Optimización de Apache Spark

TRABAJO FIN DE GRADO

CURSO 2020/21

UNIVERSIDAD COMPLUTENSE MADRID

FACULTAD DE CIENCIAS MATEMÁTICAS

GRADO EN MATEMÁTICAS. ITINERARIO DE CIENCIAS DE LA COMPUTACIÓN

Janira Ortiz Hernández

Tutores:
Carlos Gregorio Rodríguez
Luis Llana Díaz

Glosario de términos

- **Clúster:** sistema de procesamiento paralelo o distribuido que consiste en un conjunto de computadoras independientes, interconectadas entre sí, de forma que funcionan como un sólo mecanismo computacional.
- **Nodo:** cada uno de los elementos (máquinas) que componen el clúster.
- **API:** Interfaz de Programación de Aplicaciones. Es un conjunto de definiciones y subrutinas que permiten conectar varias aplicaciones y facilitar el uso de servicios entre ellas.
- **Script:** secuencia de comandos u órdenes que controlan el comportamiento de un programa.
- **Shell:** intérprete de órdenes o de comandos que proporciona una interfaz para el acceso a los servicios del sistema operativo (ejemplo: escritorio de Windows).

Resumen

Trabajando con Spark a menudo se presenta que el rendimiento obtenido es peor de lo esperado y son muchas las variables a tener en cuenta para abordarlo: una inadecuada configuración del clúster, tener un número correcto de particiones de los datos a procesar, la presencia de sesgo de datos, el formato de datos con los que se trabaja, el trabajo en memoria, la cantidad de datos que se transfieren entre las máquinas, el orden de procesamiento de datos o el tipo serialización de datos. Todas estas consideraciones son importantes a la hora de estudiar el rendimiento de nuestras aplicaciones. Profundizar en todas ellas sería un objetivo demasiado amplio para el tamaño y propósito de un Trabajo Fin de Grado, por lo que nos centraremos en los puntos que han resultado más interesantes y adecuados al tipo de aplicaciones con las que se suele trabajar.

En la primera parte del trabajo, se ha desarrollado un estudio teórico de la herramienta Spark: su arquitectura y componentes, la aplicación de sus transformaciones y operaciones definidas sobre RDDs y Dataframes, las herramientas de monitorización de las ejecuciones, la instalación y el uso de PYSPARK. Además, hay una introducción a las ejecuciones en modo clúster. La segunda parte recoge el desarrollo del proyecto propiamente dicho. Se ha basado en la creación y ajuste de distintas aplicaciones, su ejecución en el clúster y la recogida de los datos resultantes sobre ciertos puntos de interés: configuración del clúster (asignación de los recursos), particionado de datos, persistencia de los datos, sesgo de datos, control del shuffle y el trabajo con más de un dataset.

Cada punto de estudio tiene sus propios *scripts* de ejecución pero comparten de un flujo común. Este flujo nace con un primer *script* en el que se seleccionan los datasets y las aplicaciones que, junto al resto de características de la ejecución pasadas como argumento (punto de interés, número de repeticiones por prueba...), generan la llamada a un segundo *script* encargado de generar los archivos donde se guardarán los resultados. Además, este *script* realiza las llamadas a la ejecución de las aplicaciones y la recogida la información útil contenida en los archivos de log que produce Spark.

Las ejecuciones se han desarrollado bajo las condiciones de un clúster de 6 nodos, de 4 núcleos y 8GB de RAM cada uno. Esto implica que los resultados obtenidos no son directamente trasladables a clústers con recursos diferentes. Hay que tener en cuenta los límites que se presentan según las características del entorno de trabajo del que dispongamos. A pesar de todo, las conclusiones generales nos aportarán un mayor conocimiento de cómo abordar estos límites y aplicar los conocimientos adquiridos.

Abstract

This paper aims to analyze some of the performance optimization aspects of Apache Spark applications. When working with Spark, it often happens that the performance obtained is worse than expected and there are many variables to take into account to address it: an inadequate cluster configuration, having the correct number of partitions of the data to be processed, the presence of data bias, the data format with which we work, the work in memory, the amount of data transferred between machines, the order of data processing or the type of data serialization. All these considerations are important when studying the performance of our applications. To go into all of them in depth would be too broad an objective for the size and purpose of a Final Degree Project, so we will focus on the points that have been most interesting and appropriate to the type of applications with which we usually work.

In the first part of the work, a theoretical study of the Spark tool has been developed: its architecture and components, the application of its transformations and operations defined on RDDs and Dataframes, the tools for monitoring executions, the installation and use of PYSPARK, and an introduction to cluster mode executions. The second part covers the development of the project itself and is based on the creation and adjustment of different applications, their execution on the cluster and the collection of the resulting data on six points of interest: cluster configuration (resource allocation), data partitioning, data persistence, data bias, shuffle control and working with more than one dataset.

Each study point has its own execution scripts but they share a common flow. This flow starts with a first script in which the datasets and applications are selected which, together with the rest of the characteristics of the execution passed as arguments (point of interest, number of repetitions per test...), generate the call to a second script in charge of generating the files where the results will be saved and, in this order, make the calls to the execution of the applications and the collection of the useful information contained in the log files produced by Spark.

The executions have been developed under the conditions of a cluster of 6 nodes, 4 cores and 8GB of RAM each. This implies that the results obtained are not directly transferable to clusters with different resources. It is necessary to take into account the limits that are presented according to the characteristics of the working environment we have, although the general conclusions will provide us with a better knowledge of how to deal with these limits and apply the knowledge acquired.

Índice general

Capítulo 1

Introducción

Apache Spark es una de las herramientas más populares del mercado para el trabajo relacionado con Big Data. Se trata de un motor de procesamiento de datos distribuidos, creado con la intención de mejorar aspectos deficientes de su antecesor, Hadoop-MapReduce. Spark tiene grandes cualidades que le hacen ser en la actualidad la opción de uso elegida en grandes empresas y, como consecuencia, existen una gran demanda de profesionales que manejen esta herramienta en el mercado.

Este trabajo contiene esencialmente dos partes diferenciadas: una más teórica , que se desarrollará en los capítulos 2 y 3, y que se encarga de presentar el marco conceptual y técnico de la programación paralela en clústers de computadoras; la segunda parte es aplicada y se basa en la realización y análisis de pruebas de ejecución que son presentados en el capítulo 4.

Tras un análisis previo de múltiples aspectos que podrían influir en la eficiencia y la velocidad de ejecución de aplicaciones de Apache Spark, hemos decidido centrar nuestro estudio en los siguientes puntos clave:

1. Trabajar con dos tipos distintos de datasets, ambos de texto.
2. Configuración del clúster: asignación de los recursos en cuanto al número de nodos y las características de éstos.
3. Particionado de datos: elección del número de particiones teniendo en cuenta la configuración del punto anterior.
4. Persistencia de los datos para el control de la evaluación perezosa.
5. Sesgo de datos: identificación de la presencia de sesgo en nuestros datos y planificación a la hora de abordarlo.
6. Control del *shuffle* (mezclado de datos entre particiones).

1.1. Objetivos

Objetivo general:

- Analizar la capacidad de optimización del rendimiento y de los recursos en las aplicaciones Spark en diferentes aspectos de interés sobre este marco de procesamiento.

Objetivos específicos:

- Comprender el funcionamiento interno de Apache Spark mediante el estudio teórico y la customización de la herramienta.
- Orientar en las buenas prácticas a la hora de afinar las características presentes en las aplicaciones Spark.

1.2. Planificación

El desarrollo de este trabajo se divide en dos partes principales: la exposición teórica de Apache Spark y PySpark, y la elaboración y ejecución de las aplicaciones apropiadas para abordar los puntos de interés establecidos, estudiando los resultados obtenidos.

El contenido teórico comienza contextualizando la herramienta dentro del Big Data y del ecosistema Hadoop (capítulo 2). En el capítulo siguiente, se expone toda la estructura, funcionamiento interno, características principales y todo el proceso de creación, utilización, monitorización y ejecución de las aplicaciones. En las tres primeras secciones del capítulo, se mencionan todos sus componentes, aunque no profundizará en ellos, ya que están muy presentes en la ingeniería y ciencia de datos actual. También se enumeran las mejoras que presenta respecto a la técnica de procesamiento de Hadoop (MapReduce), y se explica la estructura de trabajo y procesos internos que se desarrollan durante la vida de una aplicación. En la cuarta sección se explica uno de los conceptos más importantes para entender la lógica de Spark, la evaluación perezosa. Esta característica es determinante en la velocidad de procesamiento. En las dos siguientes secciones se detallan las estructuras de datos con las que podemos trabajar y sus tipos de operaciones, aportando ejemplos prácticos. Para terminar este capítulo, las secciones 7 y 8 abordan la visualización y ejecución de aplicaciones, que son los dos apartados, junto con la elaboración del flujo de ejecución y contenido de las aplicaciones, los apartados en los que más tiempo se ha invertido durante el desarrollo de este trabajo.

Finalizada la parte teórica, pasamos al capítulo 4 donde se aplicará gran parte del contenido anteriormente expuesto. Primero se presentan los elementos con los que vamos a preparar la metodología y contenido de las pruebas. Estos son los conjuntos de datos que se procesarán, la parte principal de los códigos que se van a ejecutar, el flujo de ejecución de las pruebas completas (incluyendo la recopilación y extracción de resultados) y ejemplos de la recogida de los resultados. Esto abarcará las cuatro primeras secciones. En la quinta sección se comienzan a exponer resultados, con lo que empezamos a acotar y dirigir las pruebas en cuanto a los datasets y las aplicaciones. A partir de aquí, abordaremos los conceptos y métodos que se han fijado como objetivo de análisis: sesgo de datos, ajuste de la configuración de los recursos del clúster, el correcto particionado de los datos, control del tamaño del *shuffle*, el uso de la persistencia de los datos intermedios y las diferencias al trabajar con diferentes tipos de datos. A la hora de afrontar cada uno de estos puntos se seguirá el siguiente esquema de desarrollo:

- Exponer la problemática que conlleva y ejemplos de estas situaciones: ¿En qué afecta a mi trabajo?, es decir, ¿cómo pierdo rendimiento en las aplicaciones por esta razón?
- Estudiar las soluciones existentes, intentando proponer alguna mejora.
- Tomar datos y aplicaciones concretas en las que se dé la circunstancia y trabajar las soluciones y sus variantes para sacar conclusiones: ¿Qué solución ha dado un mejor resultado?,¿si hago ajustes de la solución original obtengo un mejor rendimiento en la ejecución?
- Pruebas con distintos tipos de datos: ¿el tipo de datos afecta al rendimiento en este sentido?

Finalmente se encuentra la sección 4.10 donde, si aplica, se realizará un análisis sobre la aplicación de más de una de las mejoras alcanzadas en una misma aplicación. El objetivo será comprobar si la acumulación de técnicas contribuye o no al rendimiento de las aplicaciones.

Para completar esta memoria, se expondrán las conclusiones e interpretaciones respecto a los resultados obtenidos, basándonos en el contenido de este trabajo y en las investigaciones que hayan podido surgir frente a resultados no esperados o no contemplados.

Para el almacenamiento y actualización de los scripts de ejecución se hará uso de la plataforma GitHub: `https://github.com/JaniraOrtizH/TFG_ApacheSparkOptimization`

Capítulo 2

Contexto

2.1. Marco conceptual

Nos remontamos a los años cuarenta para identificar la primera referencia a un problema que iba a gobernar una parte significativa de nuestra evolución social: la explotación de la información. En 1944, el bibliotecario de la Universidad de Wesleyan, Fremont Rider, realizó una estimación muy contundente. Aseguraba que la recopilación de información que se llevaba desarrollando duplicaba su tamaño cada dieciséis años. Por tanto, para el año 2040 la biblioteca de la Universidad de Yale tendría alrededor de 200.000.000 volúmenes en su poder. En las siguientes décadas, con el nacimiento de la memoria virtual y el procesamiento de datos modernos, comenzaría un viaje de crecimiento exponencial de la información y, con ello, el estudio y el almacenamiento masivo de datos. Fue en 1997 cuando se escuchó hablar por primera vez del problema del Big Data por parte de dos investigadores de la NASA. En consonancia con la visión de Fremont Rider, afirmaban que el ritmo de incremento de datos comenzaba a ser una problemática para los sistemas informáticos de los que se disponía.

Aunque no hay una definición estándar, definiremos Big Data como los datos con un volumen, una diversidad y una complejidad tales que necesitan nuevas técnicas, algoritmos y arquitecturas para procesarlos. A día de hoy, cuando hablamos de Big Data nos referimos no solo a un concepto, si no a un abanico de ideas y tecnologías que han llegado a nuestros tiempos por necesidades, sobre todo, empresariales, siendo parte de la causa y del efecto del avance tecnológico. Las redes sociales, las transacciones bancarias, las investigaciones tecnológicas o las redes de sensores son ejemplos de generación masiva de contenido que resulta interesante de almacenar y procesar. Un objetivo de gran motivación para el estudio de estos datos es el de predecir eventos futuros en base a los ocurridos.

Hasta 2005 los avances en velocidad de procesamiento se centraban en la mejora de hardware, requiriendo cambios de desarrollo para la mejora de las aplicaciones. Fueron los límites de los componentes electrónicos respecto a la disipación del calor los que frenaron este avance en el procesamiento individual y se comenzó a agregar núcleos de CPU que trabajarían a la misma velocidad y de forma paralela. Pero su nacimiento como objetivo de trabajo se atribuye a Google y se sitúa con la publicación, en 2003, del estudio *The Google file system* presentado en la *ACM SIGOPS Operating Systems Review*. En este estudio se habla del sistema de ficheros distribuidos Google File System, pieza fundamental para desarrollar el trabajo de procesamiento de información en la nube y del procesamiento paralelo.

Un año más tarde, también por parte de Google, se divulgaban los detalles de lo que sería el punto de partida para el comienzo de la "era Big Data": el esquema de programación MapReduce. Este paradigma, que detallaremos más adelante, fue adoptado por el proyecto de código abierto Apache Hadoop y es la base en la que está construido Spark, que es parte del ecosistema Hadoop y nuestro objeto de estudio. Fue Matei Zaharia, en 2009 como proyecto dentro de AMPLab en la Universisdad de Berkeley, quién realizó el primer desarrollo de Spark. En 2013 fue donado a Apache Software Foundation convirtiéndolo en un proyecto de alto nivel. Actualmente Apache Spark es uno de los proyectos *open source* más activos

y la comunidad de código abierto más grande de Big Data, con más de mil colaboradores, algunos muy importantes, como IBM, Intel, Databricks, Cloudera o Facebook. Esto se debe a su gran potencia y robustez, la gran mejora en velocidad respecto a las ejecuciones en Hadoop, el acceso asequible a usuarios que no posean niveles avanzados en MapReduce o Java y, sin duda, la escalabilidad. Por todo esto, una gran cantidad de proveedores de tecnología han apostado por este servicio, integrando esta tecnología a sus servicios y colaborando activamente con su desarrollo y mejora.

2.2. Marco teórico

2.2.1. Las matemáticas en el Big Data

La explotación de datos en la que vivimos y cuyo crecimiento está garantizado debido a la gran demanda por parte del mercado, requiere en gran medida del uso de técnicas y algoritmos matemáticos, tanto para el procesado como en el análisis de esta ingente cantidad de información.

Comenzando con el planteamiento de nuestros objetivos de estudio, será importante el análisis descriptivo de la información de la que disponemos. La estadística nos proporciona formas de descripción fundamentales para la presentación de nuestros datos y su posterior análisis. Una vez tenemos nuestros datos organizados y bien interpretados, es esencial establecer las dependencias entre las variables que se nos presenten. Expresar estas relaciones como ecuaciones nos permite hacer uso de álgebra lineal, facilitando los problemas que requieren de operaciones en espacios multidimensionales. La programación lineal y la optimización lineal también jugarán papeles de gran peso en trabajos de ciencia de datos.

Centrando la atención en la parte computacional de los trabajos con Big Data, es obvio que las matemáticas tienen una gran presencia. Todos los algoritmos con los que se trabaja en el desarrollo del procesado de datos han sido diseñados y reflexionados por humanos, basándose en ecuaciones y modelos matemáticos. La era del Big Data está fundamentada sobre pilares matemáticos y su avance está fuertemente ligado a los modelos y algoritmos matemáticos.

2.2.2. Apache Hadoop

Apache Hadoop fue el primer ecosistema open source Big Data. Fue desarrollado por Yahoo! en 2005 permitiendo el trabajo con clústers. Inspirado en el estudio de GFS de Google y el algoritmo de programación MapReduce, nació como una solución para grandes volúmenes de datos tanto estructurados como no-estructurados difíciles de procesar con las tecnologías de las que se disponía hasta el momento.

Hadoop se considera un ecosistema ya que engloba una serie de servicios Big Data, como son la ingesta, el almacenamiento, el análisis y el sostenimiento. En este ecosistema encontramos servicios muy demandados actualmente como Hbase, Hive, Sqoop, Kafka o Spark, entre otros.

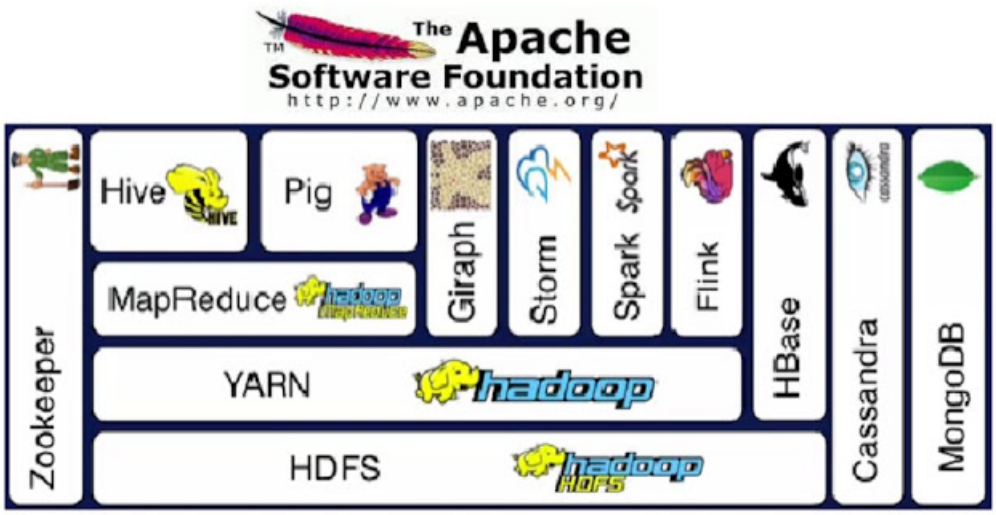


Figura 2.1: Ecosistema Hadoop. Recuperado de http://www.apache.org

Este motor de procesamiento tiene cuatro componentes fundamentales:

- Hadoop Common: librerías de Java y scripts para el inicio de Hadoop y que sirven de apoyo a otros módulos de éste.
- Yarn: una estructura de planificación de tareas y administración de los archivos que se encuentran en la distribución del clúster.
- HDFS: sistema de archivos distribuidos altamente tolerante a fallos ya que está diseñado para almacenar de manera confiable en conjuntos de máquinas. Los archivos son almacenados por lotes, todos ellos del mismo tamaño excepto el último, y son replicados en varios de los nodos con el fin de cubrir las posibles pérdidas de archivos por falta de disponibilidad de alguna de las máquinas.
- MapReduce: paradigma de programación que proporciona un esquema de ejecución paralela entre múltiples nodos. Esta versión moderna de "divide y vencerás" trabaja internamente a través de dos etapas centrales: map y reduce, definidas con respecto a datos con una estructura de tipo clave-valor (tuplas). Además, existe una fase intermedia llamada *shuffle* que se encarga de ordenar por claves los resultados proporcionados por la fase de mapeo y combinar en una lista todos los valores asociados a una clave.

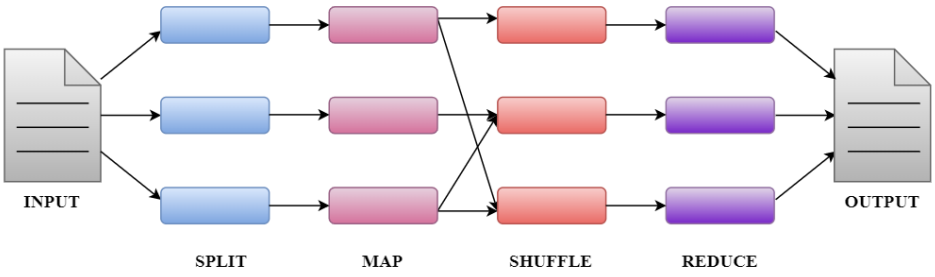


Figura 2.2: Esquema MapReduce

La función *map* definida es aplicada en paralelo para cada ítem en la entrada de datos previamente particionados en pedazos llamados *splits* (el entorno Hadoop se encarga de este particionado), generando una lista de pares por cada partición. La función *reduce* realiza una llamada para cada clave única y algún tipo de fusión en la lista de valores asociados para producir un conjunto más

pequeño. De esta forma se obtiene un esquema de trabajo pensado para el procesamiento en clúster, distribuyendo la labor entre los nodos existentes.

Hadoop tiene una estructura de maestro/esclavo, donde el *Namenode* (el maestro) hace las funciones de árbitro y repositorio de los metadatos de HDFS. Los *Datanodes* (generalmente uno por nodo) se responsabilizan de la creación, eliminación y replicación de los bloques de datos, siempre bajo las instrucciones del *Namenode*.

2.3. Marco de desarrollo

Para el desarrollo del siguiente trabajo haremos uso de una serie de herramientas de hardware y software, algunas personales y otras proporcionadas por la Universidad Complutense de Madrid. La labor de uso y búsqueda de información, desarrollo de la memoria, construcción de los programas y conexión remota al clúster se ha realizado en un ordenador portátil marca Lenovo y modelo Yoga 700-14isk, con las siguientes características técnicas: procesador Intel Core i7, disco duro 256GB SSD SATA III y 8GB de memoria RAM. Como sistema operativo se ha hecho uso tanto de Ubuntu 16.04 como de Windows 10. Windows instalado de forma principal y Ubuntu a través de una máquina virtual con el software de virtualización para arquitecturas x86/amd64 Oracle VM VirtualBox.

El clúster utilizado, llamado Dana, se trata de un conjunto de 6 CPUs (nodos) con entorno Linux y 8 GB de RAM y 4 núcleos por nodo. Consta de las siguientes versiones de las herramientas utilizadas: Spark 2.3.0, Python 3.5.3 y Hadoop 2.7.2.

Para una mayor accesibilidad desde el ordenador personal a los datos generados en Dana durante las ejecuciones se ha hecho uso de la herramienta MobaXterm, que proporciona un terminal para Windows con commandos de linux.

Capítulo 3

Apache Spark

Apache Spark se puede definir como una plataforma de computación en clúster desarrollada para ser rápida y de propósito general. En esencia, es un motor computacional que se ocupa de programar, distribuir y monitorizar aplicaciones que constan de múltiples tareas en un clúster.

3.1. Componentes

Dispone de componentes preparados para la interoperación y para manejar varias cargas de trabajo, como SQL o MLLIB. Esto permite la combinación de dichos componentes como bibliotecas de un proyecto de software. La unificación del software implica una optimización en costo de ejecución y la posibilidad de crear aplicaciones polivalentes que solo requieren el mantenimiento de un sistema. En la siguiente figura vemos estos componentes. La parte inferior muestra las opciones existentes para la gerencia del clúster: Standalone, YARN o Mesos. El resto de módulos corresponden a los componentes principales en la estructura de Spark.

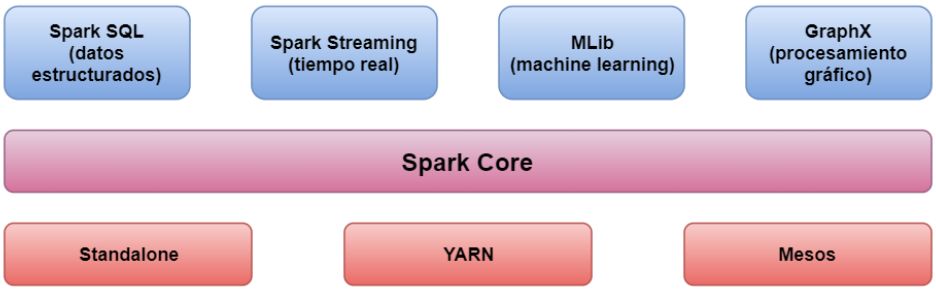


Figura 3.1: Componentes principales de Spark

El Spark Core contiene las funcionalidades básicas de programación de tareas, gestión de memoria, recuperación de fallos, interacción con sistemas de almacenamiento y alberga la API que define la estructura de datos principal de Spark: los conjuntos de datos distribuidos resistentes (RDDs).

Spark SQL es el componente que permite el trabajo con datos estructurados mediante consulta de tipo SQL, combinado con el resto de operaciones soportadas por los conjuntos de datos.

Spark Streaming proporciona una extensión de Spark Core para el procesamiento de datos en mini-batches, simulando el procesado en tiempo real.

MLlib es el paquete para el trabajo en *machine learning* y algoritmos estadísticos. Incluye algoritmos de clasificación, regresión, arboles de decisión, mínimos cuadrados, agrupación, etc.

Por último, GraphX ofrece una API para la computación gráfica. Proporciona una manera eficiente de visualizar, transformar o unir gráficos con RDDs.

En el desarrollo de este trabajo nos centramos en programas escritos con Spark Core, aunque también se realizará alguna ejecución haciendo uso de Spark SQL.

3.2. Apache Hadoop vs Apache Spark

Spark es un motor informático y un conjunto de librerías para el procesamiento paralelo de grandes volúmenes de datos. Está diseñado como *framework* montado en un clúster de computación para el trabajo con Big Data. Se considera la evolución lógica de Hadoop, el cual presenta ciertas limitaciones en el aprovechamiento de la programación distribuida como en el trabajo con procesos iterativos y procesos que requieren varios pasos intermedios. Esto es debido al costoso almacenamiento en disco. Así, el procesamiento en memoria de Spark para los resultados intermedios le otorga una velocidad de procesamiento de datos cien veces mayor que la de su antecesor. Esta característica disminuye las operaciones de lectura y escritura de datos cuyo coste de almacenamiento en disco puede ser muy elevado.

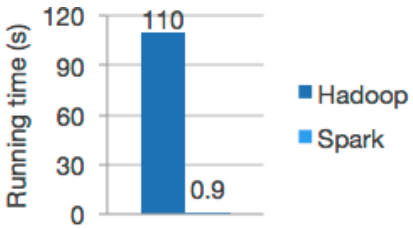


Figura 3.2: Regresión logística en Hadoop y Spark. Recuperado de spark.apache.org

Otra gran cualidad de Spark es su generalidad. Dispone de APIs nativas para los lenguajes Scala (lenguaje en que está diseñado), Java, Python y R, las cuales proporcionan un estilo de programación más sencillo. Puede ejecutarse tanto de forma local en los sistemas operativos Windows, Linux y OS X, como en Hadoop, lo que brinda una gran interconexión con servicios del ecosistema Hadoop.

A pesar de que su enfoque no es el almacenamiento de datos, se presentan gran cantidad de tipos de almacenamiento soportados por este motor de análisis. Sistemas de almacenamiento distribuido como HDFS, tuberías de datos en tiempo real, sistemas de almacenamiento persistentes, bases de datos relacionales o no y muchos otros. Otras características importantes que ofrece Spark y que veremos durante el desarrollo del trabajo son: tolerancia a fallos, reusabilidad, evaluación perezosa, soporte para análisis y eficiencia.

Podemos destacar varios aspectos en la comparación entre Hadoop MapReduce y Spark:

MapReduce	Apache Spark
Requiere la ejecución con Hadoop	Puede ejecutarse de forma independiente a Hadoop
Procesado en Batch	Procesado en Batch y Real Time
Bajo coste	Mayor coste económico (RAM)
Usa el replicado de datos para la tolerancia a fallos	Además del replicado de datos, usa los RRDs y otros modelos de almacenaje para la tolerancia a fallos
No contempla la eliminación de duplicados en los datos	Procesa los datos introducidos exactamente una vez, eliminando la posibilidad de duplicación
Lenguajes soportados: Java, C, C++, Python, Perl y Groovy	Lenguajes soportados: Java, Scala, Python y R
Manejo complejo	Manejo más sencillo gracias a la variedad de APIs que presenta
Soporta consultas de tipo SQL a través de HIVE	Tiene su propio módulo de consulta SQL: Spark SQL
Depende de un planificador externo	Tiene su propio planificador

Tabla 3.1: Hadoop vs Spark

3.3. Arquitectura

Un clúster agrupa los recursos de varias máquinas juntas. Esto posibilita el uso de estos recursos como si de una sola máquina se tratase mediante un *framework* que coordine el trabajo entre los nodos. Ésta es la función principal de Spark.

Podemos ejecutar Spark de dos formas:

- Modo local: ejecución de aplicaciones en el sistema operativo, orientado al desarrollo y las pruebas de los programas.
- Entorno distribuido: ejecución en clúster, donde la gestión de recursos es muy importante y se requiere un sistema de administración para ello. Contamos con los modos Standalone, YARN y Mesos.

Como mencionamos en la sección anterior, para la gestión del clúster existen administradores de recursos o *cluster managers* como Standalone, YARN o Mesos. Estos gerentes se encargan de coordinar los procesos independientes de los que se componen las aplicaciones ejecutadas en clúster y distribuir los ejecutores de Spark a través del sistema distribuido según los parámetros de configuración establecidos. No es conveniente utilizar Standalone para clústers grandes debido a la sobrecarga de ejecutar los servicios de Spark en los nodos del clúster y a que solo soporta aplicaciones Spark. Por el contrario, YARN y Mesos son los administradores recomendados para la ejecución en clústers de producción grandes. No producen sobrecarga de los servicios y son de propósito general, soportando diversas cargas de trabajo propias del ecosistema Hadoop y Apache.

Una aplicación de Spark consiste en dos procesos fundamentales. El principal es el proceso controlador o *driver process*, el cual ejecuta el programa controlador (*main*) en un nodo del clúster y proporciona la información relevante de la aplicación durante el tiempo de ejecución. Además, analiza, distribuye y programa el trabajo entre el resto de los procesos. Estos procesos "secundarios" son los procesos que adquiere Spark en los nodos del clúster, los ejecutores o *executor processes*. Son los encargados de realizar las tareas que requiera el *driver*, almacenar datos para la aplicación e informar sobre el desarrollo de

la misma. Puede haber más de un ejecutor por nodo en el clúster pero un ejecutor no puede residir en más de un nodo. La estructura que se acaba de describir se denomina estructura maestro/esclavo (*driver/executors*).

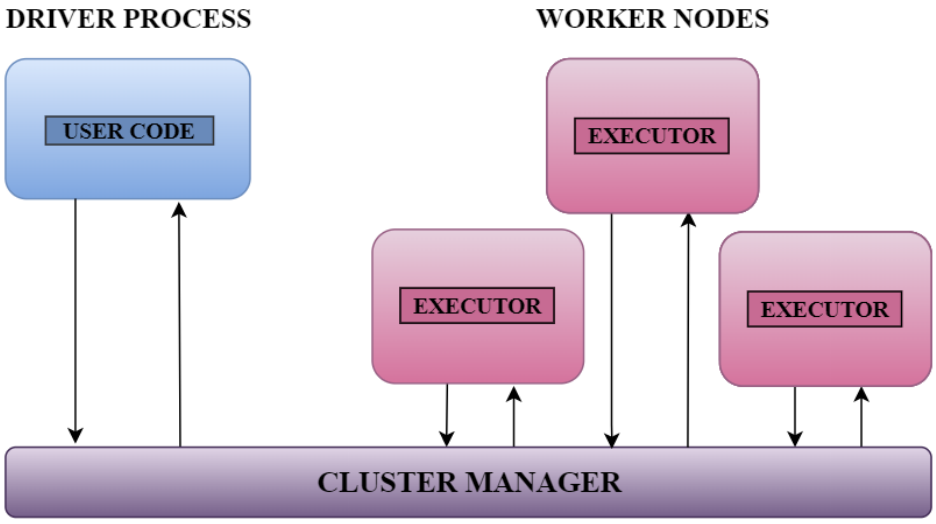


Figura 3.3: Estructura de Spark en clúster

Se dispone de soportes interactivos para el análisis de datos *ad hoc* similares a otras *shells* como pueden ser las propias de R, Python y Scala, o el *bash* para operaciones de sistema.
Nos centraremos en el uso de programas independientes (scripts). A diferencia trabajando en el *shell*, al trabajar con aplicaciones independientes tenemos que inicializar el contexto y la configuración de Spark: *SparkConf* y *SparkContext*. El objeto *SparkContext* se puede entender como la configuración que se determina para la ejecución de la aplicación de Spark. La aplicación nace en el *driver* con la definición de una serie de trabajos por parte del *SparkContext*, que también se encarga de determinar los recursos que se asignan a los ejecutores. Los parámetros de configuración de los objetos *SparkContext* son almacenados por el *SparkConf*. Algunos de estos parámetros establecen propiedades de la aplicación y otros se utilizan para asignar recursos en el clúster, como el tamaño de memoria y núcleos disponibles por ejecutor. Una vez creado el objeto *SparkConf*, es inmutable para la aplicación en cuestión.

Configuración	Descripción	Ejemplo
set(key: String, value: String)	Establece una variable de configuración	SparkConf() .set("spark.executor .instances", 4)
setAppName(String)	Establece el nombre de la aplicación	SparkConf() .setAppName("My app")
setMaster(String)	Establece la URL maestra de conexión	SparkConf() .setMaster("local")

Tabla 3.2: Hadoop vs Spark

Toda esta configuración también se puede realizar de forma externa al código, mediante los argumentos de configuración del *spark-submit* que veremos en la sección 3.8

Existen otras estructuras específicas para diversos procesados de datos: *StreamingContext* para el trabajo en *streaming*, SqlContext para la recogida o transformación de datos mediante sentencias SQL o HiveContext para trabajar con el servicio Hive de Hadoop. A partir de la versión 2.0 se presentó un

nuevo punto de entrada para poder trabajar con DataSets y DataFrames: SparkSession. Engloba todos los contextos mencionados anteriormente, incluyendo el *SparkContext*.

3.4. Evaluación perezosa

Las estructuras de datos propias de Spark (RDDs, DataFrames, DataSets) son inmutables, es decir, no se puede alterar una vez creadas. Pero podemos crear otra estructura a partir de "modificaciones" sobre otras dando instrucciones específicas a Spark de cómo queremos hacerlo. Serán modificaciones abstractas, no ejecutará hasta que no se efectúe una acción. Esta es una de las características más importantes de Spark: la evaluación perezosa o *lazy evaluation*. Esto significa que cuando se llama a una transformación ésta no se ejecuta en el momento, sino que se registra en un plan de actuación llamado DAG (Grafo Acíclico Dirigido) a la espera de que el *driver* pida los resultados.

La evaluación perezosa presenta varias ventajas en el ámbito de trabajo para el que está dirigido Spark:

- Permite estructurar los programas en operaciones más pequeñas reduciendo la cantidad de traspaso de datos al agrupar operaciones.
- Ahorro en computación: solo los valores necesarios se computan por lo que se omiten cálculos innecesarios.
- Reduce la complejidad en tiempo y espacio ya que no se ejecutan todas las operaciones y la acción se ejecuta solo cuando se requieren los datos.

3.5. Estructuras de datos en Spark

Veamos ahora cada una de las estructuras que nos ofrece Spark para el trabajo y manipulación de los datos:

RDDs

Un RDD o conjunto de datos distribuido resilientes es la principal abstracción de Spark para trabajar con datos. Prácticamente todo está construido en base a RDDs en Spark. Estas estructuras son colecciones de elementos tolerantes a fallos, es decir, si uno de los nodos que esté procesando una parte del RDD falla o es demasiado lento, ese proceso se volverá a lanzar en otro nodo para evitar que el proceso global falle. Cada RDD es dividido en diversas particiones que serán tratadas en diferentes nodos del clúster como tareas o *tasks* de los ejecutores. Esto es, una tarea será una operación sobre una partición del RDD, por lo que se trabajará con tantas particiones a la vez como número de tareas puedan ejecutarse en paralelo.

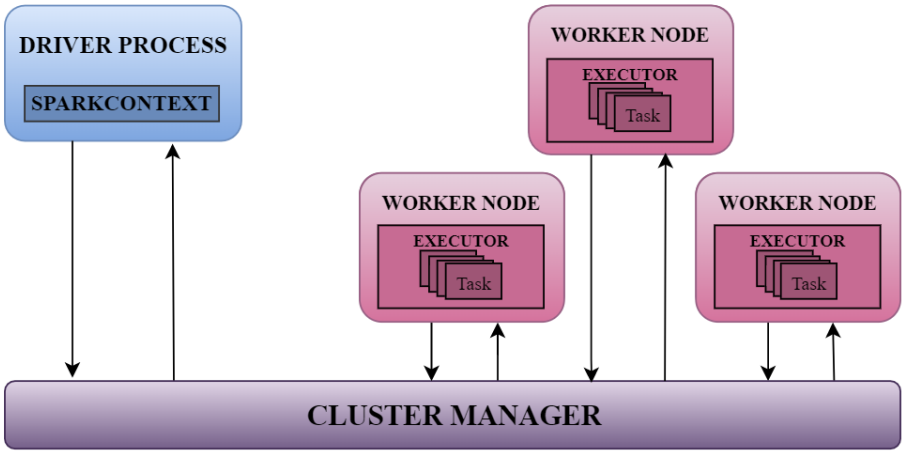


Figura 3.4: Paralelismo de tareas en los ejecutores

Podemos crear RDDs de las siguientes maneras:

- Paralelizar una colección de datos existente en el programa controlador. Aquí incluimos tanto la lectura de archivos haciendo uso de *SparkContext.textFile()*, como la distribución de colecciones de objetos como listas a través del método *SparkContext.parallelize()*.
- Hacer referencia a un conjunto de datos localizado en un sistema de almacenamiento externo, como puede ser HDFS o HBASE, que ofrezca un formato compatible con Hadoop.
- Transformando RDDs existentes mediante algunas de las operaciones que veremos en la sección 3.6

El trabajo en Spark con esta estructura de datos se resumirá en la creación de nuevas RDDs, la transformación de las ya existentes y la invocación de acciones sobre ellas.

A continuación, vamos a ver ejemplos de distintas creaciones de RDDs:

- Creación de un RDD a partir de un archivo de texto local pasado por parámetro en la ejecución del *script*:

```
from pyspark import SparkContext

sc = SparkContext()
rdd = sc.textFile('texto.txt')
```

- Creación de un RDD a partir de una colección de datos en una lista:

```
from pyspark import SparkContext

sc = SparkContext()
data = [1, 2, 3, 4, 5]
rdd = sc.parallelize(data)
```

- Creación de un RDD de pares (clave, valor) a partir de una colección de datos en una lista:

```
from pyspark import SparkContext

sc = SparkContext()
data = [('a',1), ('b',1), ('c',1), ('a',1)]
rdd = sc.parallelize(data)
```

Los RDDs de pares clave/valor son los utilizados comúnmente y son aquellos que contienen pares de elementos vinculados entre sí. En estas tuplas, la clave es el identificador y el valor es un dato asociado a la clave. Este tipo de RDDs son una construcción muy útil si pretendemos agrupar o agregar elementos que comparten un identificador. Existen operaciones específicas para este formato y algunas de ellas podemos identificarlas por ser métodos que incluyen *ByKey* en su nombre.

Para ver la utilidad de esta configuración, supongamos que queremos realizar un recuento de las apariciones de las palabras en un texto. Primero debemos convertirlo a un RDD de pares asignando el valor 1 a cada palabra. Este valor hace las veces de contador para su posterior recuento, el cual realizaremos mediante la operación *reduceByKey* que sumará para cada clave los valores asociados a ésta:

```
text = 'Creer que un enemigo debil no puede herirnos es
creer que una chispa no puede incendiar el bosque'.split(' ')

rdd = sc.parallelize(text)
pairRdd = rdd.map(lambda x: (x.lower(), 1))
wordCount = pairRdd.reduceByKey(lambda a,b: a+b)
wordCount.collect()
```

Éste sería el RDD resultante:

```
[('creer',2), ('que',2), ('un',1), ('una',1),
('enemigo',1), ('debil',1), ('no',2), ('puede',2),
('herirnos',1), ('es',1), ('chispa',1), ('incendiar',1),
('el',1), ('bosque',1)]
```

La función pasada como parámetro en la operación *reduceByKey* para el recuento de apariciones está en el formato *lambda*. Se trata de un formato propio de Python para utilizar funciones anónimas, es decir, sin necesidad de definirlas con la sentencia *def*. Más adelante veremos otras operaciones de interés al trabajar con estas estructuras.

DATASETS

Los DataSets son conjuntos de datos distribuidos que, a diferencia de los RDDs, tienen una estructura. Estas colecciones de objetos específicos orientadas al trabajo en paralelo, se caracterizan por estar fuertemente tipadas y proporcionar una interfaz de programación orientada a objetos. Proporcionan la utilidad de los RDDs, añadiendo los beneficios del componente Spark SQL.

DATAFRAMES

los DataFrames no son otra cosa que DataSets dispuestos en columnas con nombre. Podemos entender estas colecciones como tablas propias de una base de datos relacional que soportan una serie de operaciones que proporcionan una buena técnica de optimización. El esquema de trabajo con estas estructuras es similar al expuesto para los RDDs. Veamos ejemplos específicos:

- Creación de un DataFrame a partir de un RDD:

```
# Inciamos el SparkContext y el SparkSession}
from pyspark import SparkContext, SparkSession
sc = SparkContext()
spark = SparkSession.builder. getOrCreate()

# Creamos el RDD incial
data = ['hello', 'big', 'data']
paralData = sc.parallelize(data)
```

```
9
10  """Creamos el DataFrame. Primero transformamos cada
11  elemento del RDD al tipo Row() haciendo uso de la
12  transformacion .map() y despues aplicamos la operacion
13  .toDf() sobre el RDD anterior"""
14  df = paralData.map(lambda r: Row(r)).toDF(['word'])
15
16  """Mostramos el DataFrame en pantalla con
17  la accion .show()"""
18  df.show()
19  # +--------+
20  # |    word |
21  # +--------+
22  # |   hello|
23  # |     big|
24  # |    data|
25  # +--------+
```

- Creación de un DataFrame a partir de un RDD vacío, indicando la estructura de columnas que va a poseer:

```
1   from pyspark import SparkContext, SparkSession
2   # Importamos los tipos de sql que vamos a usar en el esquema
3   from pyspark.sql.types import StringType, IntegerType,
4   StructType, StructField
5   sc = SparkContext()
6   spark = SparkSession.builder. getOrCreate()
7   """Creamos el esquema que queremos para nuestro DataFrame,
8   sus columnas"""
9   schema = StructType([StructField("word",StringType(),True),
10  StructField("count",IntegerType(),True)])
11  """Usamos la operacion createDataFrame de la SparkSession,
12  dando como argumentos un RDD vacio y el esquema anterior"""
13  wordRdd = sc.emptyRDD()
14  wordDf = spark.createDataFrame(wordRdd, schema)
15  """Mostramos el DataFrame (en este caso no contiene datos)
16  en pantalla con la accion .show()"""
17  wordDf.show()
18  # +-------+-------+
19  # |   word|  count|
20  # +-------+-------+
21  # +-------+-------+
```

- Combinando los dos ejemplos anteriores, podemos crear un DataFrame con un esquema creado previamente y con los datos albergados en un RDD:

```
1   from pyspark import SparkContext, SparkSession
2   from pyspark.sql.types import StringType, IntegerType,
3   StructType, StructField
4   sc = SparkContext()
5   spark = SparkSession.builder. getOrCreate()
6   schema = StructType([StructField("word",StringType(),True),
7   StructField("count",IntegerType(),True)])
8   data = [('hello',1), ('big',1), ('data',1)]
9   rdd = sc.parallelize(data)
10  df = spark.createDataFrame(rdd, schema)
11  df.show()
12  # +-------+-------+
13  # |   word|  count|
14  # +-------+-------+
15  # |  hello|      1|
16  # |    big|      1|
17  # |   data|      1|
18  # +-------+-------+
```

- También podemos crear DataFrames a partir de distintos archivos como CSV o JSON mediante la operación *read* de la SparkSession, por ejemplo:

```
1   df = spark.read.json("test.json")
```

3.6. Operaciones sobre RDDs y DataFrames

Vamos a ver las operaciones que podemos aplicar sobre RDDs y DataFrames. No atenderemos al caso de los DataSet ya que no haremos uso de esta estructura en el presente trabajo.
Existen dos tipos de operaciones que podemos aplicar: transformaciones y acciones. La diferencia entre ambas radica en el tipo del resultado que generan. Las transformaciones devuelven un nuevo RDD/DataFrame basado en la aplicación de una operación a otro RDD/DataFrame. Las acciones devuelven un resultado final que no será un RDD/DataFrame. Esta diferencia en el output tiene consecuencias importantes en la forma interna de trabajar de Spark. En este sentido podemos entender las transformaciones como especificaciones abstractas, no se ejecutarán las modificaciones hasta que no se efectúe una acción.

3.6.1. Transformaciones

Cuando aplicamos una transformación, o varias transformaciones anidadas, solo estamos especificando una modificación que Spark registrará, pero no ejecutará.

Tenemos dos tipos de transformaciones según el número de particiones que deban cruzarse para obtener la nueva RDD. La diferencia entre ambos tipos se basa en si únicamente necesitamos la partición de origen para obtener el output o si necesitamos datos que se alojan en otras particiones. El primer caso son las llamadas *narrow transformations* o transformaciones estrechas. Tienen una dependencia uno a uno entre particiones, es decir, no es necesario mezclar los datos de las diferentes particiones, y pueden ser aplicadas en cualquier subconjunto de los datos sin necesidad de tener ninguna información sobre el resto de las particiones del conjunto.

Como se puede observar en la figura 3.5, existen casos en los que, aunque se requieran datos de distintas particiones, la operación será de tipo *narrow*. Esto es debido a que las particiones implicadas en obtener los datos de la resultante tendrán el mismo esquema de partición y, por tanto, no se desencadenará el orden aleatorio para el cruce de dichos datos (se aplica la transformación "por separado" a cada partición, pero se agrupan los datos resultantes en una sola). Veremos en profundidad este escenario en la sección 4.8.

Las *wide transformations* o transformaciones amplias son aquellas que tienen una dependencia 1 a N entre las particiones. Esto es, las particiones de input tienen datos que contribuirán a más de una de las particiones de salida que genera la acción. Estas transformaciones implican un orden aleatorio (*shuffle*). Cuando Spark necesita intercambiar los datos de más de una partición a través del clúster para realizar una operación se produce el *shuffle*, y es necesario escribir los resultados en disco. No es el caso de las transformaciones estrechas en las que todo el trabajo se realiza en memoria.

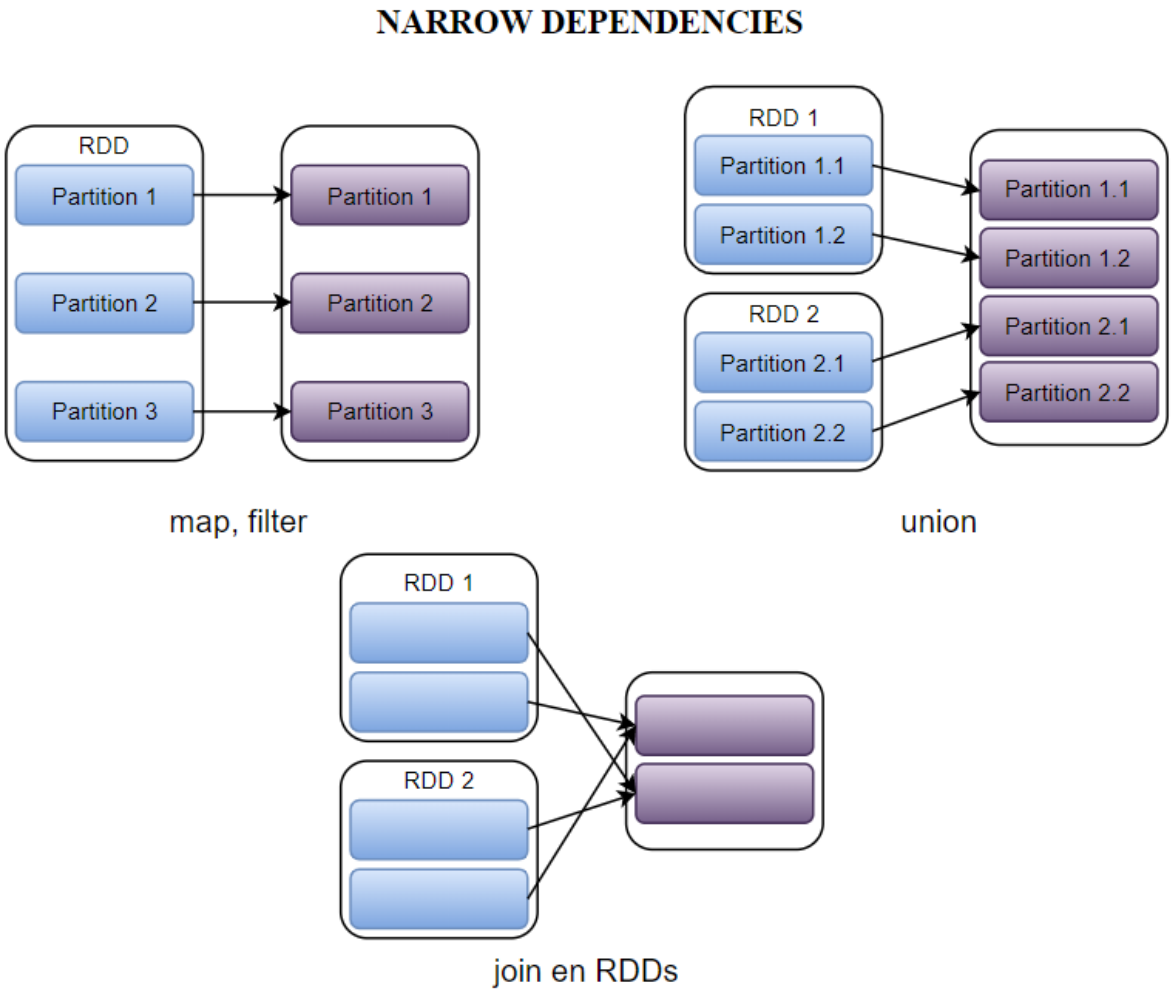


Figura 3.5: Transformaciones o dependencias *narrow*. La operación join() será de tipo *narrow* cuando ambas RDDs tengan el mismo particionador y, en consecuencia, no sea necesario el *shuffle*

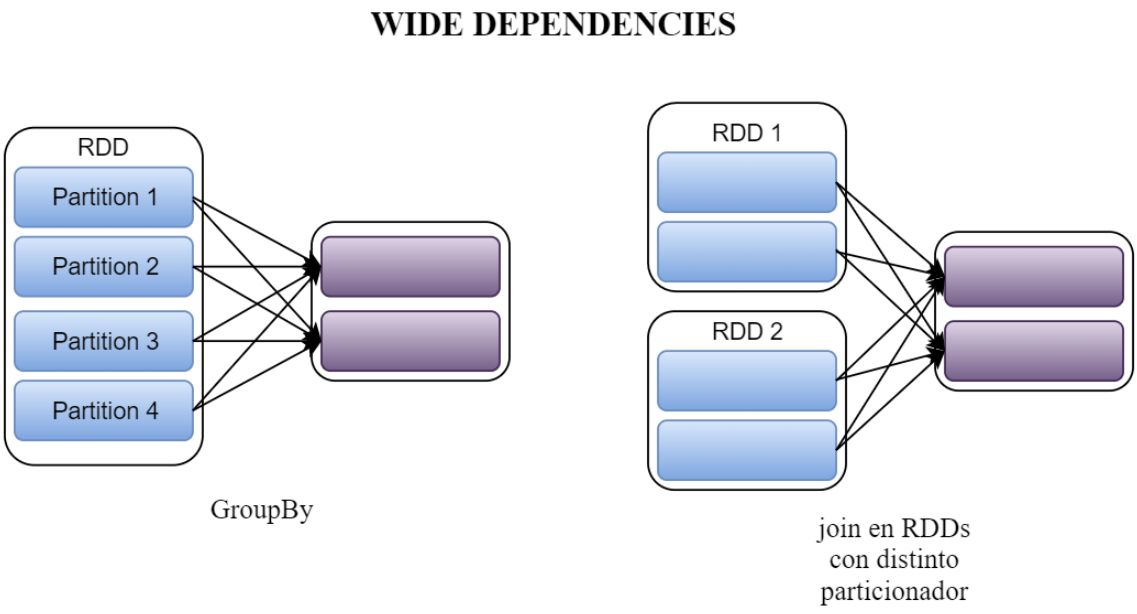


Figura 3.6: Transformaciones o dependencias *wide*

Veamos algunas operaciones de las que disponemos sobre RDDs de pares. Se plasmará el resultado mediante la acción *collect()* y las acciones glom().collect(), con la cual vemos el resultado con las separaciones correspondientes a las particiones del RDD:

Función	Descripción	Ejemplo	Ejemplo.collec()	Ejemplo.glom() .collec()
map(func)	Aplica la función argumento a cada elemento del RDD	rdd1.map(lambda x: (x[0],x[1]*2))	[('a',2),('b',2), ('c',4),('a',2)]	[[('a',2),('b',2)], [('c',4),('a',2)]]
flatMap(func)	Aplica la función argumento a cada elemento del RDD generando cero o más elementos por cada uno del RDD	rdd1.flatMap (lambda x: (x[0],x[1]*2))	[('a',2),('b',2), ('c',4),('a',2)]	[[('a',2),('b',2), ('c',4),('a',2)]]
filter(func)	Aplica un filtro al RDD según cumpla la sentencia dada como argumento	rdd1.filter(lambda x: x[1] >1)	[('c',4)]	[[], [('c',4)]]
mapPartitions (func)	Aplica la función argumento a cada partición del RDD	rdd1.mapPartitions (lambda x: (x[0], x[1]*2))	[('a',2),('b',2), ('c',4),('a',2)]	[[('a',2),('b',2)], [('c',4),('a',2)]]
union()	Produce un solo RDD producto de la unión de los elementos de dos RDDs dados	rdd1.union(rdd2)	[('a',1),('b',1), ('c',2),('a',1), ('a',3),('a',1), ('b',2),('c',1)]	[[('a',1),('b',1)], [('c',2),('a',1)], [('a',3),('a',1)], [('b',2),('c',1)]]
join()	Combina los RDDs en función de las claves. Esta transformación es de tipo*narrow* cuando ambos RDDs tienen el mismo particionador	rdd2.join(rdd3)	[('b',(2,1)), ('a',(3,1)), ('a',(1,1)), ('c',(1,1))]	[[('b',(2,1)),('c',(1,1))], [('a',(3,1)),('a',(1,1))], [], []]
reduceByKey (func)	Combina valores con la misma clave según la función argumento	rdd1.reduceByKey (lambda a,b: a+b)	[('c',2),('b',1), ('a',2)]	[[('b',1),('c', 2)],[('a', 2)]]
groupByKey()	Agrupa valores con la misma clave	rdd1.groupByKey()	[('a',[1,1]), ('b',[1]), ('c',[2])]	[[('a',[1,1]),('b',[1])], [('c',[2])]]
distinct()	Elimina los duplicados en los elementos del RDD	rdd1.distinct()	[('c',2),('a',1), ('b',1)]	[[('c',2),('a',1)], [('b',1)]]

Tabla 3.3: Transformaciones en RDDs de pares:
rdd1 = [('a',1), ('b',1), ('c',2), ('a',1)],
rdd2 = [('a',3), ('a',1), ('b',2), ('c',1)],
rdd3 = [('a',1), ('b',1), ('c',1)].
Vista de las particiones de los rdds:
rdd1 = [[('a',1), ('b',1)], [('c',2), ('a',1)]],
rdd2 = [[('a',3), ('a',1)], [('b',2), ('c',1)]],
rdd3 = [[('a',1)], [('b',1), ('c',1)]]

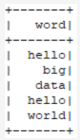
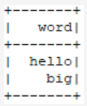
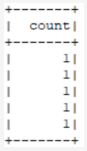
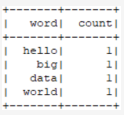
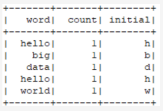
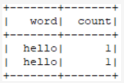
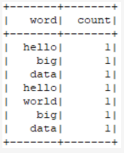
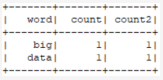
Función	Descripción	Ejemplo	Ejemplo .collec()
select(column_names)	Devuelve un DataFrame formado por el conjunto de columnas dadas por parámetro	df1.select('word')	
limit(n)	Devuelve un DataFrame con el número de registros dados por parámetro	df1.limit(2)	
drop(column_names)	Devuelve un DataFrame igual que el original pero habiendo eliminado las columnas dadas por parámetro	df1.drop('word')	
distinct()	Devuelve un DataFrame igual que el original pero habiendo eliminado los registros duplicados	df1.distinct()	
withColumn(new_colu col)	Devuelve un DataFrame resultado de añadir una nueva columna al original	df1.withColumn ('initial',substring (col('word'), 1,1))	
filter(condition)	Devuelve un DataFrame resultado de aplicar la condición dada por parámetro	df1.filter(col('word') == 'hello')	
union(otherDf)	Devuelve un DataFrame resultado de la unión de dos DataFrames	df1.union(df2)	
join(otherDf)	Devuelve un DataFrame resultado de la selección de los registros coincidentes, sobre las columnas deseadas, de ambos DataFrames	df1.join(df2, on=['word'])	

Tabla 3.4: Transformaciones sobre los DataFrames:

df1

word	count
hello	1
big	1
data	1
hello	1
world	1

df2

word	count2
big	1
data	1

3.6.2. Acciones

Este tipo de operaciones, a diferencia de las transformaciones, no devuelven un nuevo RDD/DataFrame sino algún otro tipo de valor. Como hemos visto, debido a la evaluación perezosa las transformaciones crean un plan de transformación lógico que Spark registra, pero no ejecuta. Las acciones fuerzan la evaluación del plan establecido para la estructura de datos sobre la que quiere actuar y envían los datos de los ejecutores al controlador. Existen tres casos para el uso de acciones:

- Mostrar datos por consola.
- Recopilar datos en objetos propios del lenguaje en el que estemos desarrollando.
- Escribir en fuentes de datos externas.

Función	Descripción	Ejemplo	Resultado
count()	Devuelve el número de elementos en el RDD	rdd1.count()	4
collect()	Devuelve el contenido del RDD en forma de lista	rdd1.collect()	[('a',1),('b',1), ('c',2),('a',1)]
first()	Devuelve el primer elemento del RDD	rdd1.first()	('a',1)
take(n)	Devuelve una lista con los n primeros elementos del RDD (take(1) equivale a first())	rdd1.take(3)	[('a',1),('b',1),('c',2)]
saveAsTextFile (path)	Escribe los elementos del RDD en un archivo de texto en el directorio pasado como argumento	rdd1.saveAsTextFile ('/home/user/result.txt')	-
countByKey()	Devuelve un diccionario de pares (clave, valor) con el recuento de las apariciones de cada clave	rdd2.countByKey()	'a':2, 'c':1, 'b':1
reduce(func)	Realiza agregaciones tomando dos argumentos y devuelve uno del mismo tipo. La función dada debe ser conmutativa y asociativa para poder ejecutarse correctamente en paralelo	rdd1.flatMap(lambda x: [x[1]]).reduce(lambda x,y: x+y)	5

Tabla 3.5: Acciones en RDD de pares de dos particiones: rdd1 = [('a',1), ('b',1), ('c',2), ('a',1)]

Función	Descripción	Ejemplo	Resultado
count()	Devuelve el número de registros que contiene el DataFrame	df1.count()	5
columns	Devuelve una lista con los nombres de las columnas del DataFrame	df1.columns	['word', 'count']
first()	Devuelve el primer registro del DataFrame	df1.first()	Row(word='hello', count=1)
printSchema()	Devuelve el esquema del DataFrame en formato de árbol	df1.printSchema()	root |– word: string (nullable = true) |– count: int (nullable = true)

Tabla 3.6: Transformaciones sobre los DataFrames:

df1

word	count
hello	1
big	1
data	1
hello	1
world	1

3.7. DAG y Spark UI

Como ya sabemos, debido a la evaluación perezosa, una aplicación de Spark no evaluará ninguna instrucción hasta que no se llame a una acción. Así, cada acción desencadenará un plan de ejecución para todas las transformaciones asociadas a la generación del RDD sobre el que aplicar dicha operación. La llamada de una acción junto con la ejecución de su plan lógico se denomina trabajo o *job* y es el elemento de mayor rango dentro de la aplicación. Así, cada *job* corresponde con la llamada a una acción por parte del *driver*. Los *jobs* están compuestos por etapas o *stages*, que son el conjunto de operaciones que no requieren de datos albergados en otros ejecutores (que no implican *shuffle*). De esta forma, cada *stage* es el conjunto de transformaciones *narrow* previas a una transformación *wide*, que será la que genere la dependencia de *shuffle*. Por último, se encuentran las tareas o *tasks* que constituyen cada *stage*. Son los elementos de menor rango dentro de una aplicación y tendremos tantas simultáneas como cálculos paralelos se puedan ejecutar. En una misma etapa todas las tareas ejecutarán las mismas operaciones en diferentes particiones de los datos.

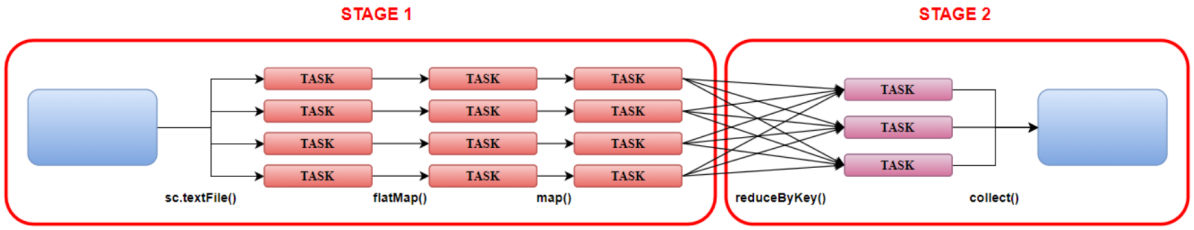


Figura 3.7: Esquema del *job* generado por la llamada a la acción *collect()* en la aplicación Word Count

El plan de ejecución que genera cada *job* se representa con un DAG que ilustra las dependencias entre las particiones del RDD. Los vértices del grafo representan los RDDs y las aristas representan las operaciones que se les aplican. Los vértices hoja (de grado uno) serán aquellos que devuelven algo diferente a un RDD, esto es, los generados por una acción y que, por lo tanto, no pueden tener hijos. De esta forma, tendremos un solo vértice de estas características por cada DAG, por cada *job*, por cada llamada a una acción.

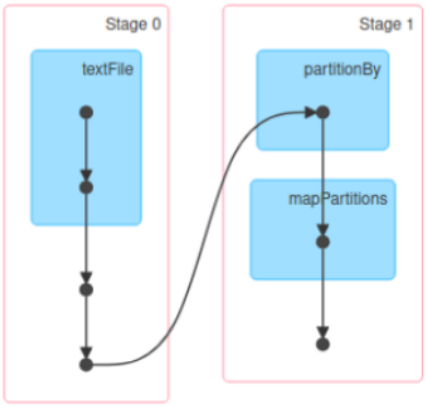


Figura 3.8: Visualización del DAG generado la aplicación Word Count

Todas estas piezas del planificador de una aplicación se puede monitorizar a través de la Consola Web de Spark, la *Application UI*, que es la interfaz web que proporciona Spark para el seguimiento de las ejecuciones. Encontramos esta interfaz de usuario en http://<driver-node>:4040. Nos ofrece acceso a distintas secciones: *Jobs* , *Stages*, *Storage*, *Enviorement* y *Executors*.

El apartado*Jobs* muestra el estado de todos los *jobs* que se han generado con nuestra aplicación, pudiendo ser activo, completado o fallido. También podemos acceder a los detalles de cada uno de los *jobs*, viendo el DAG correspondiente, el número y estado de sus *Stages* y *Tasks* o la línea temporal de todos estos eventos.

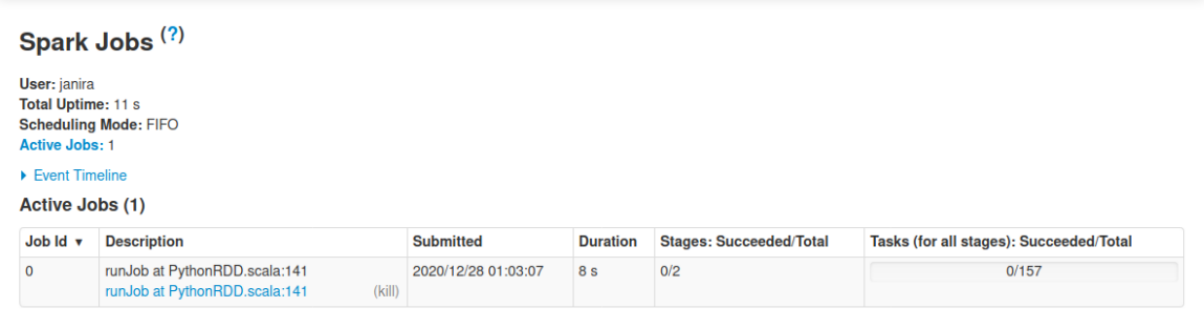


Figura 3.9: Apartado*Jobs* en Spark UI

Figura 3.10: Detalles de un Job en Spark UI

En la pestaña *Stages* podemos seguir el progreso de todas la etapas de los *job*s de la aplicación. Además de la visualización del DAG, se muestran los detalles de duración mínima, media y máxima a varios niveles: de la propia etapa, de la lectura de los bloques de *shuffle* y de la recolección de basura (GC). Estos detalles también se muestran para las tareas de las que se compone la etapa que estemos examinando.

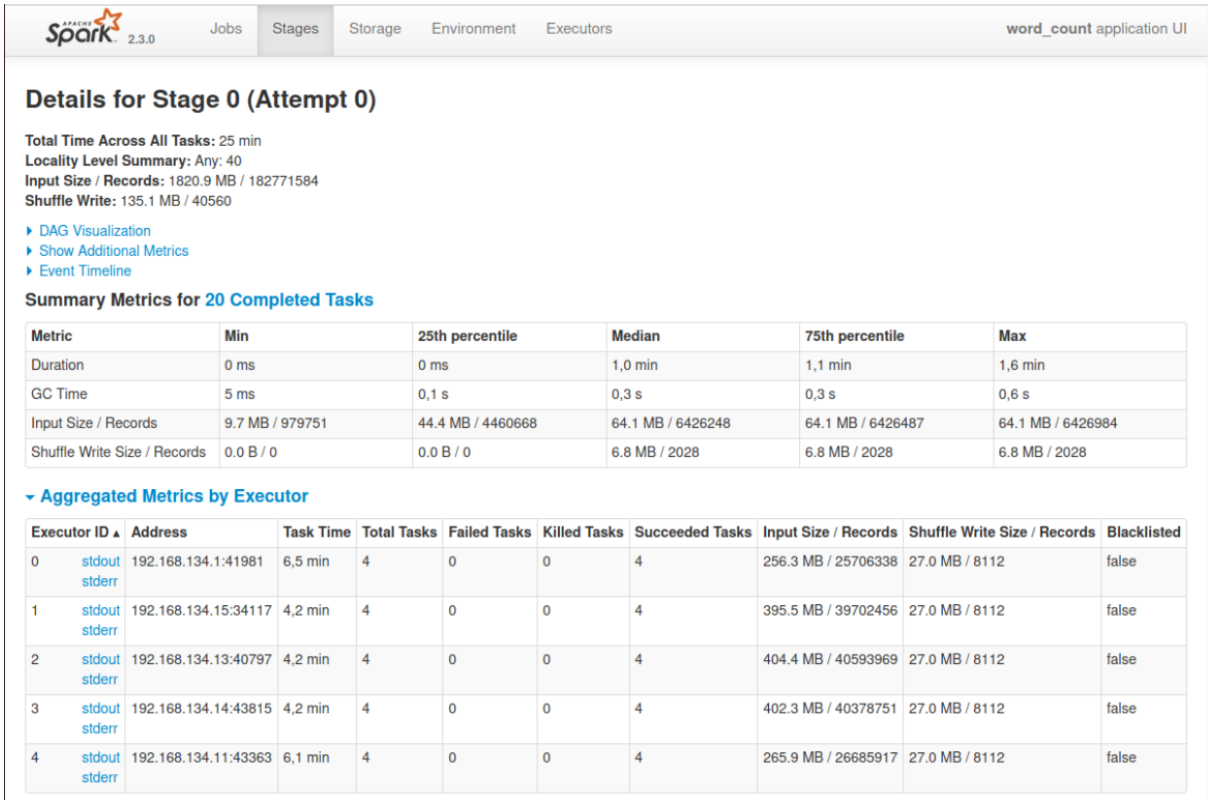


Figura 3.11: Detalles de un Stage en Spark UI

En *Storage* accedemos a la información sobre los datos cacheados en nuestra aplicación, *Environment* muestra todo lo relacionado con la configuración y propiedades de la ejecución y, por último, en el apartado *Executors* podemos ver los detalles de los ejecutores involucrados.

3.8. Spark en Python. PYSPARK

Durante el desarrollo de las pruebas utilizaremos la API de Python para Spark: PySpark. Una particularidad de esta API de Python respecto a la de Scala es que los RDDs pueden contener objetos de varios tipos, ya que Python es un lenguaje de programación dinámicamente tipado. Es decir, establecen el tipo de una variable cada vez que se le asigna un valor. Los RDDs soportan las mismas operaciones que en la API de Scala pero trabajando sobre funciones y colecciones propias de Python. Las funciones se pueden pasar como métodos de RDDs mediante la sintaxis lambda o definiendo funciones de la forma habitual y después pasándolas como argumento de los métodos de las RDDs. En los siguientes dos ejemplos vemos estas dos formas de aplicar funciones sobre los elementos de un RDD:

```
# Sintaxis lambda
data = sc.textFile(textFile)

errors = data.filter(lambda line: "ERROR" in line)
```

```
# Definir funciones
def findError(line):
  return "ERROR" in line

errors = data.filter(findError)
```

3.8.1. Instalación

Para la correcta instalación de los requisitos previos y del propio motor, aconsejo seguir los paso detallados en el siguiente enlace: `https://www.hackdeploy.com/apache-spark-installation-guide-on-ubuntu-16-04-l`
Para verificar la correcta instalación de Spark ejecutaremos en la terminal el comando *spark-shell*, tras lo que deberíamos observar en pantalla:

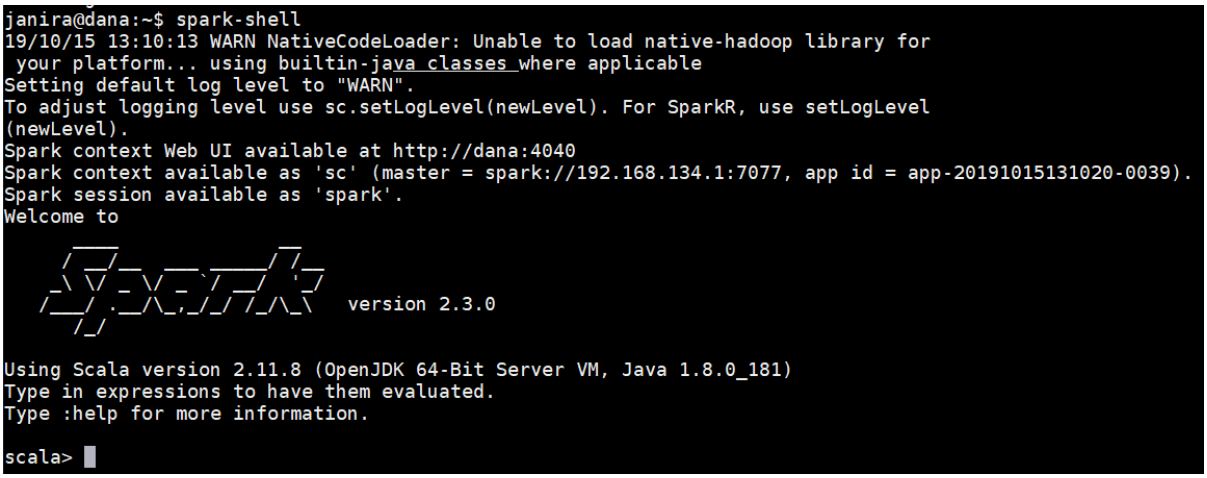


Figura 3.12: Inicialización *spark-shell*

Ahora tenemos correctamente instalado Spark en nuestra máquina Ubuntu. Como se puede observar, la *shell* que hemos ejecutado usa Scala ya que es el lenguaje en el que está basado Spark. Por último, desde el terminal y situados en el directorio donde hemos instalado Spark, ejecutaremos el comando PySpark para acceder a la API de Python:

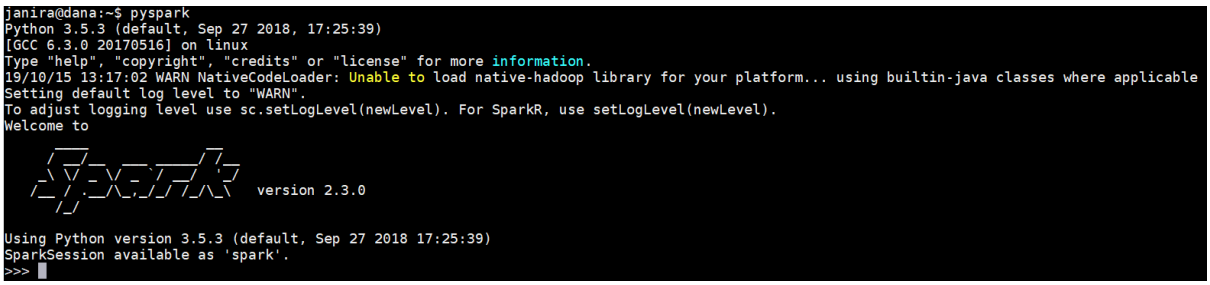


Figura 3.13: Inicialización pyspark *shell*

3.8.2. Inicialización

Trabajando a través de la *shell* se nos proporciona el objeto *SparkContext* ya inicializado. Por defecto se denomina *spark* y solo habrá que llamar a los métodos que nos proporciona para empezar a trabajar. Sin embargo, desarrollando aplicaciones mediante *scripts* necesitamos importar los métodos *SparkConf* y *SparkContext* para acceder a ellos. Nuestras primeras líneas de código deberán tener esta estructura:

```
1 from pyspark import SparkContext, SparkConf
2
3 conf = SparkConf().setAppName("First Word Count").setMaster("spark://dana:7077")
4 .set('spark.executor.cores', '4').set('spark.executor.instances', '5')
5
6 sc = SparkContext(conf=conf)
```

Figura 3.14: Inicialización de un *script* en Python

Construimos nuestra configuración en el *SparkConf* - en este caso determinamos el *master*, el nombre de la aplicación, el número de núcleos por ejecutor y el número de ejecutores – y se lo damos como argumento al *SparkContext* para que establezca dichas características. Ahora tenemos nuestro contexto inicializado, pudiendo importar nuestros datos como RDDs y aplicar las operaciones disponibles.

Veamos el ejemplo arquetipo para la programación distribuida, el programa *Word Count*, que lee un texto y realiza un recuento de las apariciones de las palabras que contiene. Pasamos el texto con el que vamos a trabajar como parámetro para el programa. Previamente debemos copiar nuestro archivo de texto desde el sistema de archivos local al sistema de archivos HDFS mediante el comando *–put*. Así, estos dos pasos desde línea de comandos serán:

```
janira@dana:~$ hdfs dfs -put quijote.txt
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/opt/hadoop/hadoop-2.7.2/share/hadoop/common/lib/slf4j-log4j12-1.7.10.j
ar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/opt/tez/tez-0.8.2/lib/slf4j-log4j12-1.7.10.jar!/org/slf4j/impl/StaticL
oggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
```

Figura 3.15: Terminal de linux. Comando para depositar los datos en el sistema de archivos HDFS

Nuestro código constará de siguiente estructura: importación de los paquetes, construcción de la configuración y el contexto, y aplicación de las operaciones básicas de este programa (*flatMap()*, *map()* y *reduceByKey()*). Además, antes de mostrar el resultado con la acción *collect()*, añadimos un filtro por la cantidad de apariciones para reducir los resultados mostrados:

```
1 from pyspark import SparkContext, SparkConf
2 import sys
3
4 conf = SparkConf().setAppName("First Word Count").setMaster("spark://dana:7077").set('spark
  .executor.cores', '4').set('spark.executor.instances', '5')
5
6 sc = SparkContext(conf=conf)
7
8 data = sc.textFile(sys.argv[1])
9
10 word_rdd = data.flatMap(lambda x: x.split(" "))
11 frec_rdd = word_rdd.map(lambda x: (x, 1))
12 result_rdd = frec_rdd.reduceByKey(lambda a,b: a+b)
13
14 print('Frecuencia de las palabras:', result_rdd.filter(lambda x: x[1]>300).collect())
```

Figura 3.16: Script Word Count Python

El resultado de la ejecución del *script* de la figura 3.16 será:

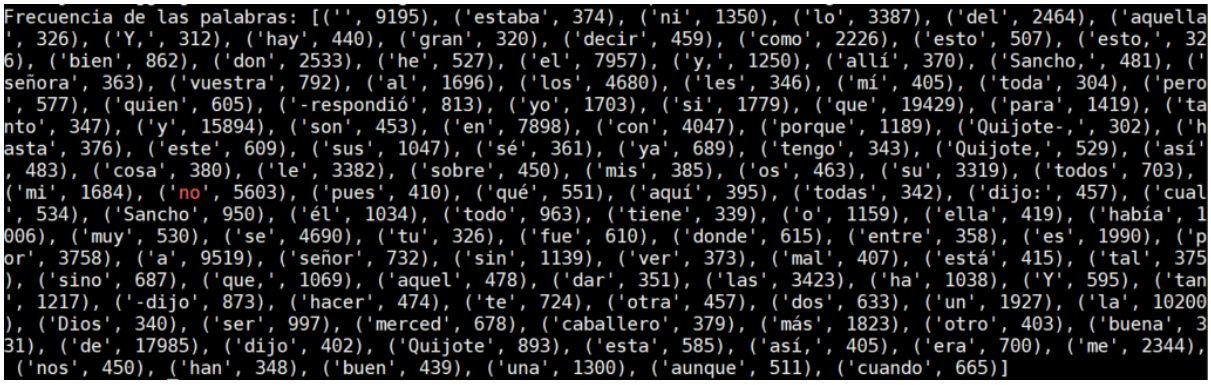


Figura 3.17: Resultado *script* Word Count Python

Como veremos, se puede refinar nuestro programa para no tener en cuenta, por ejemplo, los signos de puntuación o la distinción entre mayúsculas y minúsculas. Será cuestión de objetivo del programa que desarrollemos.

3.9. Ejecución en clúster

Para el desarrollo de las ejecuciones vamos a hacer uso de *scripts* de Python que ejecutaremos en el clúster mediante la instrucción:

```
python3 [script.py] [argumentos]
```

que envía la aplicación al *cluster manager*. Este modo de ejecución se denomina *cluster mode*. El *cluster manager* se encargará de ubicar el *driver* en uno de los nodos y los ejecutores en el resto de nodos que sean necesarios y que dispongamos. En nuestro caso haremos uso de *scripts* que no necesitan de un empaquetamiento especial, pero en caso de tener dependencias de otros proyectos se requerirá la generación de un archivo *.jar* en el que se comprimen todos los ficheros necesarios (código, dependencias, etc).

Existen otros modos de ejecución: el *client mode* y el *local mode*. El *client mode* tiene el mismo funcionamiento que el *cluster mode* con la particularidad de que el *driver* reside en una máquina externa al clúster. Mediante el *local mode* ejecutamos la aplicación de Spark en una sola máquina haciendo uso del paralelismo a través de hilos de la máquina ejecutora.

Capítulo 4

Desarrollo del proyecto

Se va a profundizar en los aspectos de Spark que se ha marcado como objeto de estudio de la optimización de nuestras aplicaciones. Para ello, se ha hecho uso de dos tipos de datasets, ambos de texto, con diferentes estructuras y se han desarrollado distintos *scripts* de ejecución según el punto de interés.

4.1. Conjuntos de datos

Los datasets que se van a procesar mediante las aplicaciones de Spark son dos tipos de archivos de texto: *words* y *dic*. Cada uno de ellos tiene una disposición:

- **dic_*.txt:** En estos archivos se almacenan diccionarios de palabras en inglés. Estas son las primera 20 líneas de texto:

  A

  A- prefix (also an- before a vowel sound) not, without (amoral). [greek]

  Aa abbr. 1 automobile association. 2 alcoholics anonymous. 3 anti-aircraft.

  Aardvark n. Mammal with a tubular snout and a long tongue, feeding on termites. [afrikaans]

  Ab- prefix off, away, from (abduct). [latin]

  Aback adv. take aback surprise, disconcert. [old english: related to *a2]

  Abacus n. (pl. -cuses) 1 frame with wires along which beads are slid for calculating. 2 archit. Flat slab on top of a capital. [latin from greek from hebrew]

  Abaft naut. —adv. In the stern half of a ship. —prep. Nearer the stern than. [from *a2, -baft: see *aft]

  Abandon —v. 1 give up. 2 forsake, desert. 3 (often foll. By to; often refl.) Yield to a passion, another's control, etc. —n. Freedom from inhibitions. abandonment n. [french: related to *ad-, *ban]

- **words_*.txt:** En este caso, el texto está compuesto por palabras, una por línea:

proclivity
stubbed
firefanged
azogrenadine
uneffectless
unfertilising
multisonous
proalcoholism
osophies
unsympathised
atherine
uncertifying
spasmotoxin
parasaboteur
seechelt
eroticizing
foiled
ignipuncture
stelliferous
quaying

Se ha escogido dos datasets de cada tipo, con tamaños muy parecidos con el objetivo de apreciar la influencia de la disposición de los datos en los tiempos de ejecución. También un dataset significativamente más grande, words_10e7_100, para observar las diferencias de resultados con el resto de datasets del mismo tipo. Así, se estará trabajando con los siguientes:

- *dic*_15.txt (1620929160 bytes ≈ 1,62 Gb)
- *dic*_30.txt (3241858320 bytes ≈ 3,24 Gb)
- *dic*_100.txt (3241858320 bytes ≈ 10,80 Gb)
- words_10e7_15 (1566368715 bytes ≈ 1,56 Gb)
- words_10e7_30 (3132737430 bytes ≈ 3,13 Gb)
- words_10e7_100 (10442458100 bytes ≈ 10,44 Gb)

4.2. Aplicaciones

En general, las ejecuciones se van a basar en 3 tipos de aplicación. Todas ellas hacen uso de una función definida como *cleanData*, encargada de eliminar los signos de puntuación presentes en el texto, transformar a minúsculas todas las letras y convertir cada línea del texto en una lista de elementos (cada palabra). De esta manera, se tendrá el texto limpio para poder realizar correctamente el conteo del número de presencias de cada palabra.

```
1 def cleanData(x):
2   x = re.sub('[^a-zA-Z0-9 ]', "", x.lower())
3   line = x.split(" ")
4   return line
```

En el caso concreto de los datasets de tipo *words* en esta función se elimina la separación de cada línea en elementos, ya que solo constará de uno, de una palabra:

```
def cleanData(x):
x = re.sub('[^a-zA-Z0-9 ]', "", x.lower())
return x
```

Ahora, se va a detallar cada una de las aplicaciones que se han desarrollado para su ejecución y la obtención de resultados:

- **Word Count:** contador clásico de apariciones de palabras en un texto:

```
data = sc.textFile(data_file)
words = data.flatMap(cleanData)
frequencies= words.filter(lambda x: x != '').map(lambda x: (x, 1)).reduceByKey(
    lambda a,b: a+b)
print('Example of words frequencies:', frequencies.collect()[1])
```

Consta de un *job* con dos *stages*. El primer *stage* realiza la limpieza de los datos y la transformación de cada elemento en un par (clave, valor), donde clave será la palabra y valor 1, que significa que hay una aparición de dicha palabra. El segundo *stage* se genera debido al barajado de datos, *shuffle*, que se realiza con la transformación *reduceByKey()*. El *job* finaliza con la acción *collect()* que recopila toda la información que contiene el RDD en una lista de pares (clave, valor), siendo el valor la suma de cada aparición de la palabra clave.

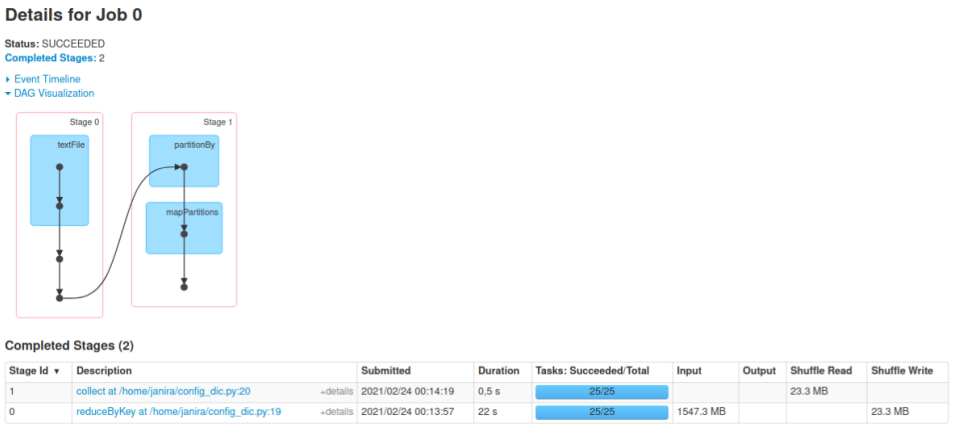


Figura 4.1: Detalles Word Count

- **Word Count Level 2:** extensión de la aplicación anterior que sustituye la clave del par por la incial de la palabra y agrupa el RDD por esta nueva clave, generando como valor el conjunto de todos los pares que tienen como inicial de palabra dicha clave. Se han desarrollado dos versiones. La primera incluye un *job* adicional con la acción *collect()* sobre el conteo clásico antes de continuar con el resto de transformaciones:

```
data = sc.textFile(data_file)
words = data.flatMap(cleanData)
frequencies= words.filter(lambda x: x != '').map(lambda x: (x, 1)).reduceByKey(
    lambda a,b: a+b)
print('Example of words frequencies:', frequencies.collect()[1])

initialsFreqs = frequencies.map(lambda x: (x[0][0], (x[0], x[1])))
groupInitialFreqs = initialsFreqs.groupByKey()
print('Example of words frequencies group by initial: ', groupInitialFreqs.
    collect()[1])
```

Figura 4.2: Detalles Word Count Level 2 con *job* adicional

La segunda extiende el *job* principal con una etapa más, es decir, no se añade la acción intermedia:

```
data = sc.textFile(data_file)
words = data.flatMap(cleanData)
frequencies= words.filter(lambda x: x != '').map(lambda x: (x, 1)).reduceByKey(
    lambda a,b: a+b)

initialsFreqs = frequencies.map(lambda x: (x[0][0], (x[0], x[1])))
groupInitialFreqs = initialsFreqs.groupByKey()
print('Example of words frequencies group by initial: ', groupInitialFreqs.
    collect()[1])
```

Figura 4.3: Detalles Word Count Level 2 con etapa adicional

- **Word Count Plus:** de nuevo, se trata de una extensión de las aplicaciones anteriores. Se añade un RDD en el que se cuentan las apariciones por inicial de palabra y, después, se realiza un *join* con el RDD que tiene como clave la inicial y como valor el conjunto de palabras con esa inicial y sus apariciones. Así, se tiene como resultado un RDD de pares de la forma (inicial, (apariciones de la inicial, conjunto de palabras con esta inicial y sus apariciones)).
También existen dos versiones, la que separa la aplicación en 3 *jobs*:

```
data = sc.textFile(data_file)
words = data.flatMap(cleanData)
frequencies= words.filter(lambda x: x != '').map(lambda x: (x, 1)).reduceByKey(
    lambda a,b: a+b)
print('Example of words frequencies:', frequencies.collect()[1])

initialsFreqs = frequencies.map(lambda x: (x[0][0], (x[0], x[1])))
groupInitialFreqs = initialsFreqs.groupByKey()
print('Example of words frequencies group by initial: ', frequencies.collect()
    [1])
```

```
9
10    reduceInitialFreqs = initialsFreqs.map(lambda x: (x[0], x[1][1])).reduceByKey(
          lambda a,b: a + b)
11    totalInfoFreqs = reduceInitialFreqs.join(initialsFreqs)
12
13    print('Example of result: ', totalInfoFreqs.collect()[1])
```

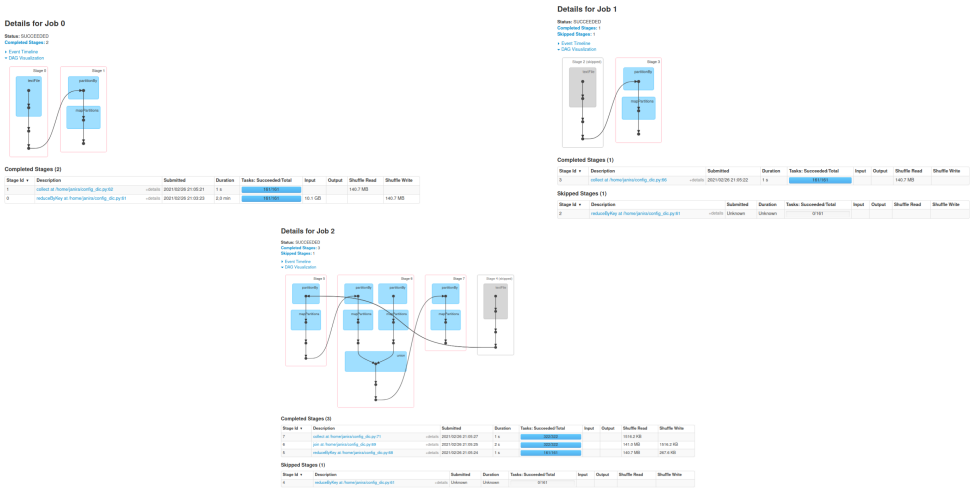


Figura 4.4: Detalles Word Count Plus con *jobs* adicionales

y la que consta de un solo *job* añadiendo etapas:

```
1     data = sc.textFile(data_file)
2     words = data.flatMap(cleanData)
3     frequencies= words.filter(lambda x: x != '').map(lambda x: (x, 1)).reduceByKey(
          lambda a,b: a+b)
4
5     initialsFreqs = frequencies.map(lambda x: (x[0][0], (x[0], x[1])))
6     groupInitialFreqs = initialsFreqs.groupByKey()
7
8     reduceInitialFreqs = initialsFreqs.map(lambda x: (x[0], x[1][1])).reduceByKey(
          lambda a,b: a + b)
9     totalInfoFreqs = reduceInitialFreqs.join(initialsFreqs)
10    print('Example of result: ', totalInfoFreqs.collect()[1])
```

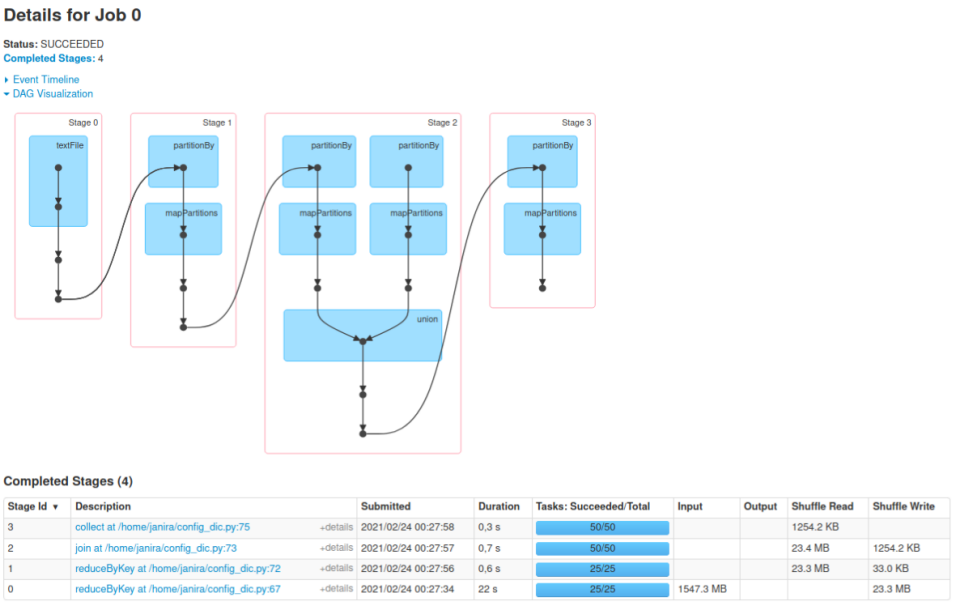


Figura 4.5: Detalles Word Count Plus con etapas adicionales

4.3. Flujo de ejecución

La ejecución de las aplicaciones expuestas en el punto anterior, se ha desarrollando mediante un conjunto de *scripts* que están asociados entre sí, mediante llamadas a ejecución en un orden concreto. Se mostrará el código del primer *script* del flujo y el resto se dejan como anexos para su consulta.

1. ***run_app.py***

```
import sys
import os

sep = sys.argv.index('+')
internal_param = sys.argv[:sep]
mode, log_dir, rep, dataset = sys.argv[sep+1:]

scriptsConfig = ['word_count','word_count_level_2','word_count_plus']

scriptsRepartition = ['word_count_repartition_n', '
    word_count_level_2_repartition_n', 'word_count_plus_repartition_n']

scriptsPersist = ['word_count_persist_mem_only', '
    word_count_level_2_pesist_mem_only', 'word_count_plus_pesist_mem_only', '
    word_count_persist_mem_and_disk', 'word_count_level_2_pesist_mem_and_disk', '
    word_count_plus_pesist_mem_and_disk', 'word_count_persist_disk_only', '
    word_count_level_2_pesist_disk_only','word_count_plus_pesist_disk_only']

scriptsSkew = ['word_count_join_vocals', 'word_count_join_vocals_broadcast', '
    word_count_join_vocals_vs_constants', 'word_count_join_vocals_vs_constants2']

allScripts = scriptsConfig + scriptsRepartition + scriptsPersist + scriptsSkew

word_files=['words_10e7_15.txt','words_10e7_30.txt','words_10e7_100.txt']
word_paths=map(lambda x: '/public_data/words/'+x, word_files)

dict_files=['dic_15.txt','dic_30.txt','dic_100.txt']
dict_paths=map(lambda x: '/public_data/words/'+x, dict_files)

data_paths={'dic':dict_paths,'words':word_paths}

apps={'config':scriptsConfig, 'repartition':scriptsRepartition, 'persist':
    scriptsPersist, 'skew':scriptsSkew, 'all':allScripts}
driver_cores = [1]
driver_mem = ['600mb','1500mb','3g']
executors = [1,2,8]
executor_mem = ['600mb','1500mb', '3g', '6g']
```

```
31    executor_cores = [2,3,4]
32    appVar = []
33
34    for data_f in data_paths[dataset]:
35      for app in apps[internal_param[1]]:
36        for r in driver_cores:
37          for s in driver_mem:
38            for t in  executor_mem:
39              for x in executors:
40                for y in executor_cores:
41                  for i in range(int(rep)):
42                    print("Repetition: "+str(i))
43                    print("Driver Cores: "+str(r))
44                    print("Driver Memory: "+str(s))
45                    print("Executors: "+str(x))
46                    print("Executor Memory: "+str(t))
47                    print("Executor Cores: "+str(y))
48                    if app in scriptsRepartition:
49                      appVar = ['3', '4', '5', '6', '7', '8', '9', '10']
50                      for n in appVar:
51                        args = str(app)+' '+str(r)+' '+str(s)+' '+str(x)+' '+str(t)+'
                               '+str(y)+' '+n+' + '+str(mode)+' '+str(log_dir)+' '+str(
                              rep)+' '+str(data_f)+' '+str(internal_param[1])
52                        os.system('python3 run_job.py '+args)
53                    elif app in scriptsPersist:
54                      appVar = ['2', '3', '4', '5']
55                      for n in appVar:
56                        args = str(app)+' '+str(r)+' '+str(s)+' '+str(x)+' '+str(t)+'
                               '+str(y)+' '+n+' + '+str(mode)+' '+str(log_dir)+' '+str(
                              rep)+' '+str(data_f)+' '+str(internal_param[1])
57                        os.system('python3 run_job.py '+args)
58                    else:
59                      args = str(app)+' '+str(r)+' '+str(s)+' '+str(x)+' '+str(t)+' '
                           +str(y)+' + '+str(mode)+' '+str(log_dir)+' '+str(rep)+' '+
                           str(data_f)+' '+str(internal_param[1])
60                      os.system('python3 run_job.py '+args)
61        if "level_2" in app:
62          os.system('python3 totals.py '+str(data_f)+' totalsLevel2.txt')
63        elif "plus" in app:
64          os.system('python3 totals.py '+str(data_f)+' totalsPlus.txt')
65        else:
66          os.system('python3 totals.py '+str(data_f)+' totalsCount.txt')
```

Este es el *script* principal donde se establecen:

- Las listas de los *scripts* a ejecutar según el punto de estudio: *scriptsConfig*, *scriptsRepartition*, *scriptsPersist*, *scriptsSkew* y *allScripts*.
- Los *paths* de cada tipo de datasets junto a los archivos de datos de ese tipo que se van a ejecutar: data_paths
- Las variables de configuración con las que se van a ejecutar las aplicaciones: driver_cores, driver_mem, executors, executor_mem, executor_cores y appVar (para los casos de ejecución de *scriptsRepartition* y *scriptsPersist*)
- Bucles de ejecución de cada aplicación con las variables de configuración anteriormente establecidas.
- Ejecución de el *script* que genera el archivo de totales por dataset.

El comando para la ejecución de este *script* tiene la siguiente estructura:

```
python3 run_app.py [apps] + [mode] [log_dir] [rep] [path]
```

donde,

- [*apps*] indica los *script* que se que van a ejecutar: *config* (estudio de la configuración del clúster), *repartition* (estudio de las particiones en los RDDs), *persist* (estudio de la persistencia de los RDDs), *skew* (estudio del sesgo de datos) o *all* (ejecutar todos los *scripts* anteriores)
- [*mode*] indica el modo de ejecución: *test* (muestra los parámetros sin realizar la ejecución) o *run* (ejecución completa)
- [*log_dir*] indica la ruta de almacenamiento de los logs de ejecución
- [*rep*] indica el número de ejecuciones por configuración que se realizarán
- [*path*] indica el tipo de dataset con el que se van a realizar las ejecuciones: *words* o *dic*.

Ejemplo: ejecución de los *script* de estudio de la configuración en modo *run*, con cuatro repeticiones por ejecución para los dataset correspondientes a *word* :

```
python3 run_app.py \textsl{config} + run /home/janira/logs 4 words
```

2. **run_job.py** Este *script* es llamado en cada vuelta de los bucles presentes en *run_app.py*. En él se generan las rutas y los archivos, según los parámetros de ejecución, para almacenar los resultados y se realiza la llamada a la función del *script* correspondiente a la aplicación que se va a ejecutar.
   Ver código completo en el anexo 5

3. **logscript.py** Script de generación del archivo *totals.txt*, en la ruta de resultados correspondiente al dataset que se procesa en la ejecución. Aquí se recogen tiempos totales y otros datos de interés sobre las ejecuciones.
   Ver código completo en el anexo 5

4. ***config*_scripts.py, *repartition*_scripts.py, persist_scripts.py, *skew*_scripts.py** Estos *scripts* recogen las diferentes funciones que corresponden a cada aplicación que se va a ejecutar según los argumentos de elección de *run_app.py*.
   Ver código completo en el anexo 5

5. **totals.py** Tras la finalización de las ejecuciones para un mismo datasets, se llama a este *script* que generará un archivo de texto *mediumTotalTimes.txt* con la media de tiempos de las repeticiones de los datos de ejecución almacenados en *totals.txt*.
   Ver código completo en el anexo 5

4.4. Recogida de resultados

Por cada *dataset*, configuración y aplicación se generarán un archivo de texto que recoge toda la información que hemos considerado relevante. Estos datos se extraen del log que Spark produce y que se trata de un archivo JSON, conteniendo toda la información relativa a las condiciones iniciales de ejecución, detalles de los *jobs* que se hayan generado y tiempos de ejecución.

Un ejemplo de los archivos que se han construido para facilitar el estudio de resultados es el siguiente. Se ha omitido la información perteneciente a las *tasks* intermedias registradas:

```
App Start Time: 2021-06-03 08:53:18.671000

  Job 0 Start, Number of Stages 2

    Stage 0 Start, Number of Tasks 49

      Finnish Task 5, Total time: 11.47
            JVM GC Time 94
            Shuffle Read Metrics {'Remote Blocks Fetched': 0, 'Local Blocks Fetched': 0, '
                 Fetch Wait Time': 0, 'Remote Bytes Read': 0, 'Remote Bytes Read To Disk': 0,
                 'Local Bytes Read': 0, 'Total Records Read': 0}
            Shuffle Records Written 539
            Shuffle Write Time 11086773
            Shuffle Bytes Written 936846
            Input Records Read 1077895
            Input Bytes Read 67174400
            Output Records Written 0
            Output Bytes Written 0

    [...]

      Finnish Task 47, Total time: 9.369
            JVM GC Time 5
            Shuffle Read Metrics {'Remote Blocks Fetched': 0, 'Local Blocks Fetched': 0, '
                 Fetch Wait Time': 0, 'Remote Bytes Read': 0, 'Remote Bytes Read To Disk': 0,
                 'Local Bytes Read': 0, 'Total Records Read': 0}
            Shuffle Records Written 539
            Shuffle Write Time 3908904
            Shuffle Bytes Written 937645
            Input Records Read 1077858
```

```
            Input Bytes Read 67174400
            Output Records Written 0
            Output Bytes Written 0


      ....................................    Stage 1 Start, Number of Tasks 49

        Finnish Task 52, Total time: 0.167
            JVM GC Time 0
            Shuffle Read Metrics {'Remote Blocks Fetched': 39, 'Local Blocks Fetched': 10, '
                Fetch Wait Time': 13, 'Remote Bytes Read': 750720, 'Remote Bytes Read To Disk
                ': 0, 'Local Bytes Read': 192138, 'Total Records Read': 539}
            Shuffle Records Written 0
            Shuffle Write Time 0
            Shuffle Bytes Written 0
            Input Records Read 0
            Input Bytes Read 0
            Output Records Written 0
            Output Bytes Written 0

      [...]

        Finnish Task 97, Total time: 0.076
            JVM GC Time 0
            Shuffle Read Metrics {'Remote Blocks Fetched': 39, 'Local Blocks Fetched': 10, '
                Fetch Wait Time': 10, 'Remote Bytes Read': 762899, 'Remote Bytes Read To Disk
                ': 0, 'Local Bytes Read': 195303, 'Total Records Read': 539}
            Shuffle Records Written 0
            Shuffle Write Time 0
            Shuffle Bytes Written 0
            Input Records Read 0
            Input Bytes Read 0
            Output Records Written 0
            Output Bytes Written 0


      ....................................  Finnish Job 0, Total time: 51.053

  App Finnish Time: 2021-06-03 08:54:12.642000

      Stage 0 completed info:
        Max task time: 17.819, task ID: 11
        Min task time: 3.018, task ID: 40
        Medium time of tasks: 11.258673469387753      Medium JVCGCTimeMetrics of tasks:
            67.79591836734694      Medium ShuffleRecordsMetrics of tasks: 539.0      Medium
            ShuffleWriteMetrics of tasks: 8339268.653061224      Medium ShuffleBytesWritten of
             tasks: 937661.775510204    Stage 1 completed info:
        Max task time: 0.182, task ID: 52
        Min task time: 0.044, task ID: 95
        Medium time of tasks: 0.08906122448979592      Medium JVCGCTimeMetrics of tasks:
            0.2857142857142857      Medium ShuffleRecordsMetrics of tasks: 0.0      Medium
            ShuffleWriteMetrics of tasks: 0.0      Medium ShuffleBytesWritten of tasks: 0.0
  App Total Time 53.971 in seconds, 0.8995166666666666
```

Además de los resultados individuales, se generan archivos que únicamente recogen los tiempos por aplicación y los tiempos medios de las *task* por dataset y por tipo de aplicación. Estos archivos se identifican como *totals* y tienen la siguiente estructura de contenido:

```
/public_data/words/dic_30.txt['word_count', 'n', '6g', '1', '6g', '3']MediumTaskTime-Stage0;12.173653061224488
/public_data/words/dic_30.txt['word_count', 'n', '6g', '1', '6g', '3']MediumTaskJVCGCTimeMetrics-Stage0;118.18367346938776
/public_data/words/dic_30.txt['word_count', 'n', '6g', '1', '6g', '3']MediumTaskShuffleRecordsMetrics-Stage0;539.0
/public_data/words/dic_30.txt['word_count', 'n', '6g', '1', '6g', '3']MediumTaskShuffleWriteMetrics-Stage0;12198918.367346939
/public_data/words/dic_30.txt['word_count', 'n', '6g', '1', '6g', '3']MediumTaskShuffleBytesWritten-Stage0;937661.775510204
/public_data/words/dic_30.txt['word_count', 'n', '6g', '1', '6g', '3']MediumTaskTime-Stage1;0.09814285714285718
/public_data/words/dic_30.txt['word_count', 'n', '6g', '1', '6g', '3']MediumTaskJVCGCTimeMetrics-Stage1;1.163265306122449
/public_data/words/dic_30.txt['word_count', 'n', '6g', '1', '6g', '3']MediumTaskShuffleRecordsMetrics-Stage1;0.0
/public_data/words/dic_30.txt['word_count', 'n', '6g', '1', '6g', '3']MediumTaskShuffleWriteMetrics-Stage1;0.0
/public_data/words/dic_30.txt['word_count', 'n', '6g', '1', '6g', '3']MediumTaskShuffleBytesWritten-Stage1;0.0
/public_data/words/dic_30.txt['word_count', 'n', '6g', '1', '6g', '3']APPTotalTime;40.515
/public_data/words/dic_30.txt['word_count', 'n', '6g', '1', '6g', '4']MediumTaskTime-Stage0;13.97773469387755
/public_data/words/dic_30.txt['word_count', 'n', '6g', '1', '6g', '4']MediumTaskJVCGCTimeMetrics-Stage0;176.73469387755102
/public_data/words/dic_30.txt['word_count', 'n', '6g', '1', '6g', '4']MediumTaskShuffleRecordsMetrics-Stage0;539.0
/public_data/words/dic_30.txt['word_count', 'n', '6g', '1', '6g', '4']MediumTaskShuffleWriteMetrics-Stage0;16430871.265306123
/public_data/words/dic_30.txt['word_count', 'n', '6g', '1', '6g', '4']MediumTaskShuffleBytesWritten-Stage0;937661.775510204
/public_data/words/dic_30.txt['word_count', 'n', '6g', '1', '6g', '4']MediumTaskTime-Stage1;0.23377551020408163
/public_data/words/dic_30.txt['word_count', 'n', '6g', '1', '6g', '4']MediumTaskJVCGCTimeMetrics-Stage1;4.571428571428571
/public_data/words/dic_30.txt['word_count', 'n', '6g', '1', '6g', '4']MediumTaskShuffleRecordsMetrics-Stage1;0.0
/public_data/words/dic_30.txt['word_count', 'n', '6g', '1', '6g', '4']MediumTaskShuffleWriteMetrics-Stage1;0.0
/public_data/words/dic_30.txt['word_count', 'n', '6g', '1', '6g', '4']MediumTaskShuffleBytesWritten-Stage1;0.0
/public_data/words/dic_30.txt['word_count', 'n', '6g', '1', '6g', '4']APPTotalTime;41.028
/public_data/words/dic_30.txt['word_count', 'n', '1500m', '1', '1500m', '3']MediumTaskTime-Stage0;12.172244897959189
/public_data/words/dic_30.txt['word_count', 'n', '1500m', '1', '1500m', '3']MediumTaskJVCGCTimeMetrics-Stage0;128.44897959183675
/public_data/words/dic_30.txt['word_count', 'n', '1500m', '1', '1500m', '3']MediumTaskShuffleRecordsMetrics-Stage0;539.0
/public_data/words/dic_30.txt['word_count', 'n', '1500m', '1', '1500m', '3']MediumTaskShuffleWriteMetrics-Stage0;11690966.530612245
/public_data/words/dic_30.txt['word_count', 'n', '1500m', '1', '1500m', '3']MediumTaskShuffleBytesWritten-Stage0;937661.775510204
/public_data/words/dic_30.txt['word_count', 'n', '1500m', '1', '1500m', '3']MediumTaskTime-Stage1;0.09134693877551023
/public_data/words/dic_30.txt['word_count', 'n', '1500m', '1', '1500m', '3']MediumTaskJVCGCTimeMetrics-Stage1;0.6122448979591837
/public_data/words/dic_30.txt['word_count', 'n', '1500m', '1', '1500m', '3']MediumTaskShuffleRecordsMetrics-Stage1;0.0
/public_data/words/dic_30.txt['word_count', 'n', '1500m', '1', '1500m', '3']MediumTaskShuffleWriteMetrics-Stage1;0.0
/public_data/words/dic_30.txt['word_count', 'n', '1500m', '1', '1500m', '3']MediumTaskShuffleBytesWritten-Stage1;0.0
/public_data/words/dic_30.txt['word_count', 'n', '1500m', '1', '1500m', '3']APPTotalTime;40.079
```

Figura 4.6: Archivo de totales para el dataset dic_30.txt, la aplicación Word Count y alguna de las configuraciones ejecutadas

Por último, se generan los archivos de totales ordenados por resultado y tiempo para mejor visualización. De estos archivos, se crean (mediante el *script* totals.py) tres por dataset, esto es, uno por cada tipo de aplicación (Word Count, Word Count Level 2 y Word Count Plus):

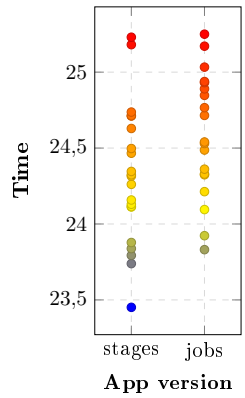
```
("/public_data/words/dic_30.txt['word_count', 'n', '1500m', '1', '1500m', '3']APPTotalTime", 40.079)
("/public_data/words/dic_30.txt['word_count', 'n', '6g', '1', '6g', '3']APPTotalTime", 40.515)
("/public_data/words/dic_30.txt['word_count', 'n', '6g', '1', '6g', '4']APPTotalTime", 41.028)
("/public_data/words/dic_30.txt['word_count', 'n', '3g', '2', '3g', '2']APPTotalTime", 43.2455)
("/public_data/words/dic_30.txt['word_count', 'n', '1500m', '1', '1500m', '4']APPTotalTime", 43.476)
("/public_data/words/dic_30.txt['word_count', 'n', '6g', '2', '3g', '2']APPTotalTime", 44.269)
("/public_data/words/dic_30.txt['word_count', 'n', '3g', '3', '3g', '2']APPTotalTime", 45.445)
("/public_data/words/dic_30.txt['word_count', 'n', '6g', '2', '6g', '2']APPTotalTime", 53.971)
("/public_data/words/dic_30.txt['word_count', 'n', '6g', '2', '6g', '2']MediumTaskTime-Stage1", 0.08906122448979592)
("/public_data/words/dic_30.txt['word_count', 'n', '1500m', '1', '1500m', '3']MediumTaskTime-Stage1", 0.09134693877551023)
("/public_data/words/dic_30.txt['word_count', 'n', '6g', '1', '6g', '3']MediumTaskTime-Stage1", 0.09814285714285718)
("/public_data/words/dic_30.txt['word_count', 'n', '6g', '1', '6g', '4']MediumTaskTime-Stage1", 0.23377551020408163)
("/public_data/words/dic_30.txt['word_count', 'n', '1500m', '1', '1500m', '4']MediumTaskTime-Stage1", 0.23473469387755097)
("/public_data/words/dic_30.txt['word_count', 'n', '3g', '3', '3g', '2']MediumTaskTime-Stage1", 0.3276734693877551)
("/public_data/words/dic_30.txt['word_count', 'n', '3g', '2', '3g', '2']MediumTaskTime-Stage1", 0.36278571428571427)
("/public_data/words/dic_30.txt['word_count', 'n', '6g', '2', '3g', '2']MediumTaskTime-Stage1", 0.39938775510204083)
("/public_data/words/dic_30.txt['word_count', 'n', '6g', '2', '6g', '2']MediumTaskTime-Stage0", 11.258673469387753)
("/public_data/words/dic_30.txt['word_count', 'n', '1500m', '1', '1500m', '3']MediumTaskTime-Stage0", 12.172244897959189)
("/public_data/words/dic_30.txt['word_count', 'n', '6g', '1', '6g', '3']MediumTaskTime-Stage0", 12.173653061224488)
("/public_data/words/dic_30.txt['word_count', 'n', '6g', '1', '6g', '4']MediumTaskTime-Stage0", 13.97773469387755)
("/public_data/words/dic_30.txt['word_count', 'n', '1500m', '1', '1500m', '4']MediumTaskTime-Stage0", 14.267346938775512)
("/public_data/words/dic_30.txt['word_count', 'n', '6g', '2', '3g', '2']MediumTaskTime-Stage0", 14.425816326530615)
("/public_data/words/dic_30.txt['word_count', 'n', '3g', '2', '3g', '2']MediumTaskTime-Stage0", 14.476948979591839)
("/public_data/words/dic_30.txt['word_count', 'n', '3g', '3', '3g', '2']MediumTaskTime-Stage0", 14.47785714285714)
```

Figura 4.7: Archivo final de totales para el dataset dic_30.txt, la aplicación Word Count y alguna de las configuraciones ejecutadas

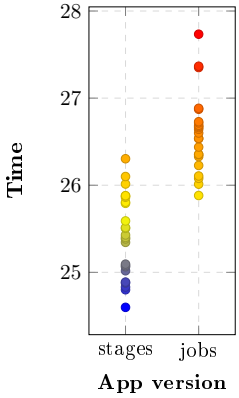
Con esta forma de recopilación de datos se permite realizar una gran cantidad de ejecuciones y poder visualizar los resultados a medida que se generan por cada dataset y tipo de aplicación. Es decir, no es necesario que terminen todas la ejecuciones para generar estos archivos. La llamada al *script* totals.py se encuentra al final de cada bloque correspondiente a un dataset y un tipo de aplicación.

4.5. Evidencias Previas

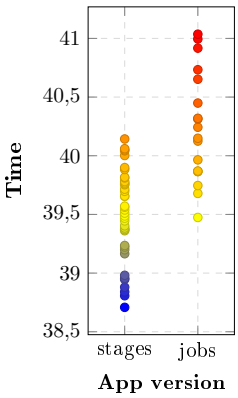
Como se ha expuesto en la sección 4.2 se han desarrollado las aplicaciones en dos versiones: un *job* y segmentación en más de un *job*. Con esta diferenciación se puede ver el impacto en tiempo de ejecución del uso de acciones de muestra de resultados intermedios. Aunque no es un diferencia muy grande, en las siguientes gráficas se puede apreciar una mayor concentración de resultados con mayor tiempo de ejecución para las aplicaciones con acciones añadidas.



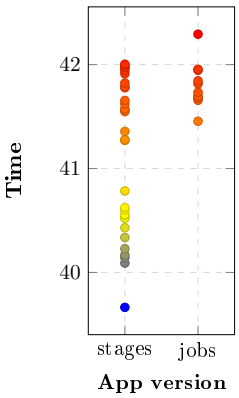
(a) dic 15 - Level 2



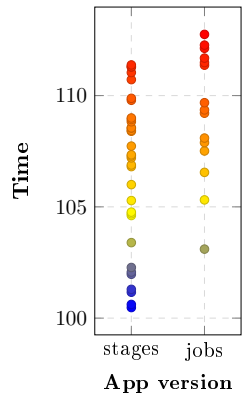
(b) dic 15 - Plus



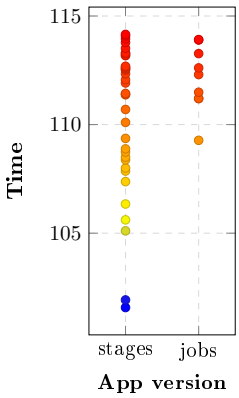
(c) dic 30 - Level 2



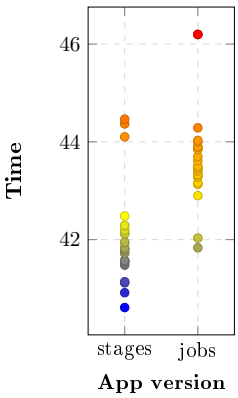
(d) dic 30 - Plus



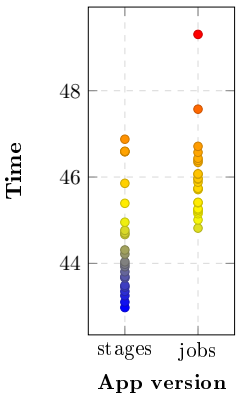
(e) dic 100 - Level 2



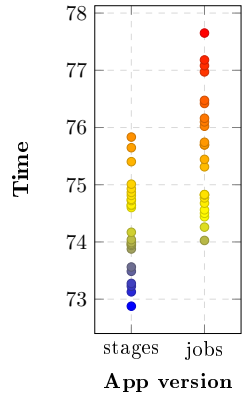
(f) dic 100 - Plus



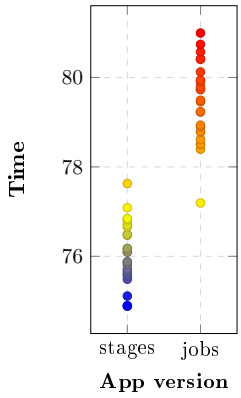
(g) Words 15 - Level 2



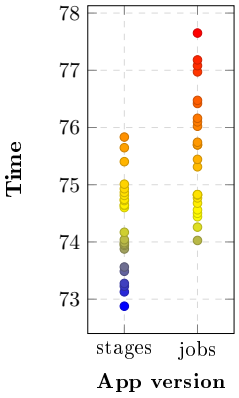
(h) Words 15 - Plus



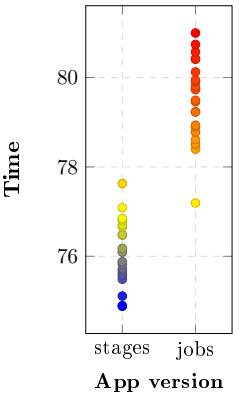
(i) Words 30 - Level 2



(j) Words 30 - Plus



(k) Words 100 - Level 2



(l) Words 100 - Plus

Figura 4.8: Gráficas de dispersión respecto a cada dataset y aplicación comparando los tiempos de las dos versiones: *stages* y *jobs*

Además, con este primer conjunto de ejecuciones para los diferentes datasets y las diferentes aplicaciones, encontramos una gran proximidad entre los tiempos de ejecución de los tres "niveles"propuestos para las aplicaciones:

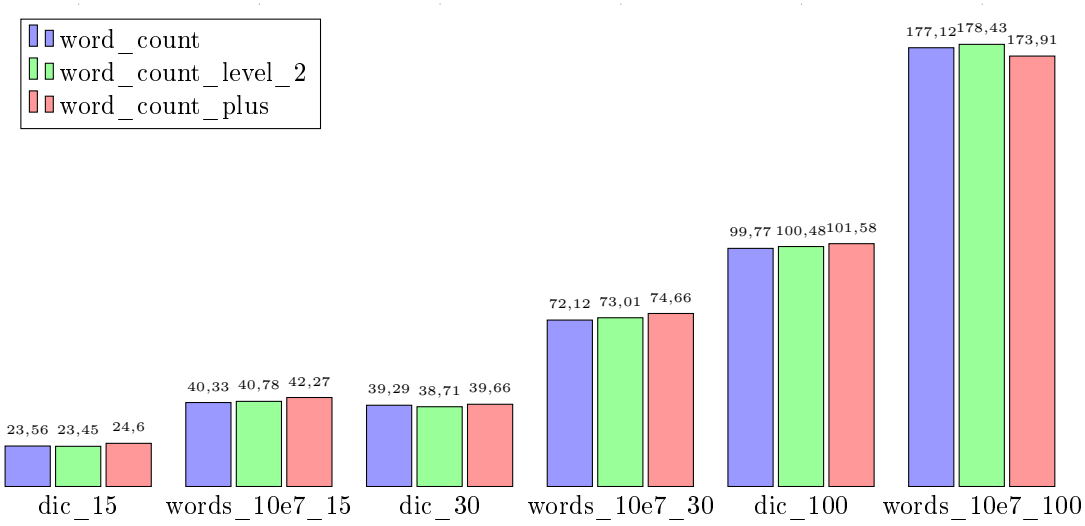


Figura 4.9: Tiempos (en segundos) de ejecución en los tres tipos de aplicación según el dataset

Aunque pequeñas, estas diferencias de tiempo se mantendrán durante todo el análisis de resultados. Por ello, no se contempla como un problema que los tiempos sean tan próximos, pues las aplicaciones tienen diferencias concretas en su contenido y merece la pena observar si en algún apartado surgen cambios respecto a la aplicación más rápida.

4.6. Sesgo de datos

Se denomina sesgo de datos a la presencia de asimetría en la distribución de nuestros datos, es decir, cuando la media, la moda y la mediana no son iguales entre sí. Es una problemática que puede conducir a una reducción de rendimiento importante. Un gran desequilibrio en la cantidad de registros asociados a las claves y, por tanto, un mal reparto de datos entre las particiones conllevará a que la tarea con más carga de datos sea más lenta en ejecución que el resto y retrase el proceso general. Esta inconsistencia en los tiempos de ejecución es obviamente ineficiente. Estamos cargando a un nodo con gran parte del trabajo general, desaprovechando capacidad de ejecución paralela en nuestros procesos.

Para realizar *joins* o agregaciones Spark necesita que los valores asociados a una clave se alberguen en una misma partición. De esta manera, si una partición resulta ser significativamente más grande que las demás, se dará el sesgo de datos. Además de tareas con sobrecarga, existen otros indicadores de que nuestros datos se presentan sesgados. El bajo uso de CPU será un indicador de que no se están aprovechando los recursos de los que se dispone. También se puede dar el caso de que la clave con sobrecarga de valores provoque un error de memoria por no caber en un ejecutor.

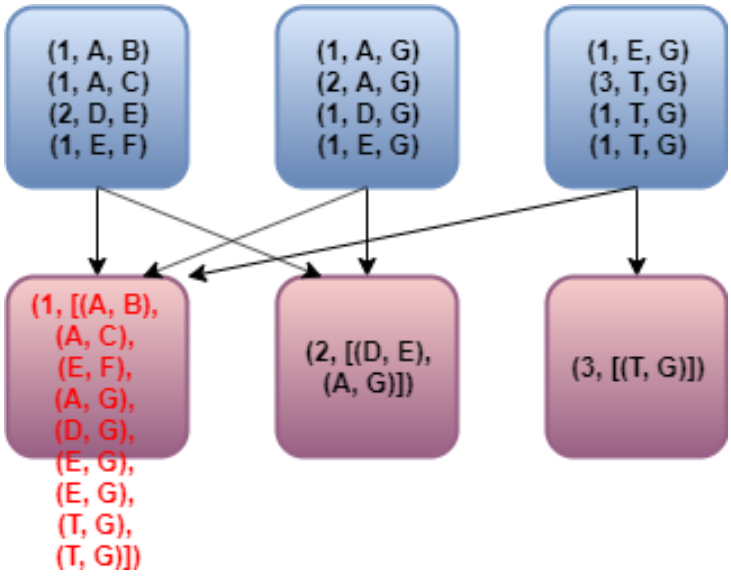


Figura 4.10: Resultado de la operación *groupByKey()* sobre datos sesgados. Se da una falta de espacio en un ejecutor para la clave desequilibrada

De las anteriores, según en la situación que se de, existen algunas posibles soluciones:

- **Trasmisión de datos (*broadcast*):** Si se va a realizar un *join* entre dos conjuntos de datos, uno de ellos presenta sesgo y el otro es lo suficientemente pequeño para guardarse en memoria en un ejecutor, se puede realizar una difusión del conjunto de datos más pequeño. De esta manera enviamos una copia de dichos datos a cada nodo y evitamos su envío en cada tarea que lo requiera. Los RDDs no son directamente transmisibles mediante el *broadcast*

- **Simplificar los *joins* asignando particionadores conocidos:** Spark tiene la capacidad de evitar el orden aleatorio en un *join* cuando una transformación anterior ha establecido el mismo particionador en ambos RDDs. Así, si se pretende transformar uno de los RDDs, deberíamos pasarle como argumento de partición el particionador del otro RDD. De esta forma, se evitará el *shuffle* en el *join* por tener el mismo particionador en ambos RDDs. Si se va a realizar una transformación en ambos y tienen el mismo número de particiones, se puede dejar el particionador *hash* por defecto ya que el particionador será el mismo. Esta estrategia se abordará en el apartado 4.8, en las pruebas realizadas sobre la aplicación **word_count_plus**.

- **Procesado previo de los datos:** Teniendo una distribución irregular de los datos presente desde origen (muchos de los registros están asociados a un pequeño número de claves), una solución inmediata sería distribuir los datos en un mayor número de claves, es decir, dividir en más de un grupo los valores de las claves superpobladas, figura 4.10.

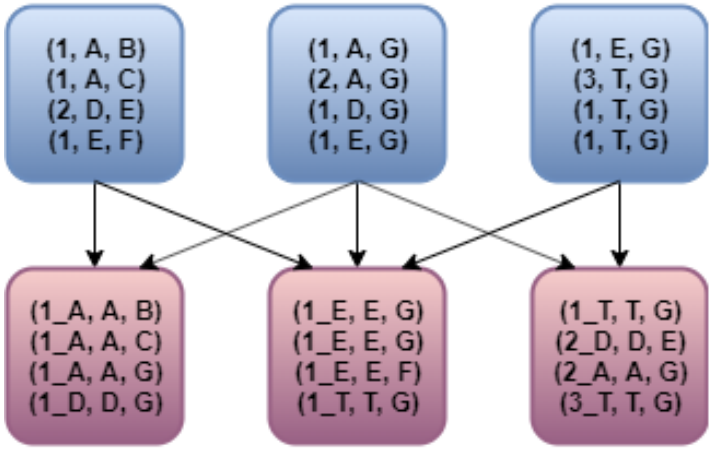


Figura 4.11: Resultado de la operación *groupByKey()* habiendo reestructurado los datos con la desagregación de la clave superpoblada

4.6.1. Resultados

Aunque se podrán observar más resultados interesantes respecto a esta sección, nos encontramos con una evidencia principal observada tras ejecutar un primer conjunto de configuraciones de prueba de las aplicaciones con los datasets propuestos. Como se ha comentado con anterioridad, la diferencia entre los datasets con los que se está trabajando es el sesgo de los datos. Para el caso del grupo **dic**, tenemos datos sesgados ya que, al contener lenguaje natural, existen palabras recurrentes que tendrán un mayor número de apariciones, como pueden ser *'a'* o *'the'*. Sin embargo, los datasets del grupo **words** no presentan este sesgo ya que la frecuencia de las palabras es uniforme.

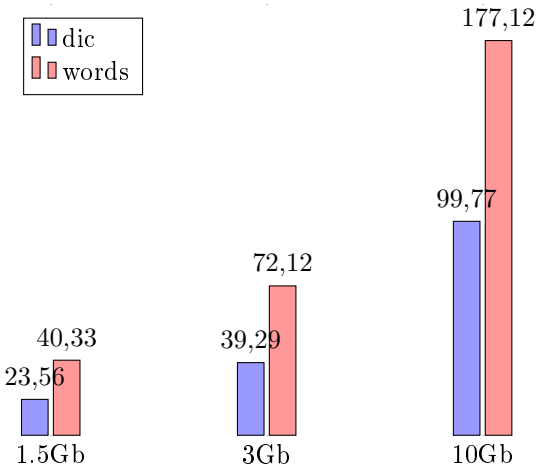


Figura 4.12: Tiempos (en segundos) de ejecución en los dos tipos de datasets según su tamaño para la aplicación correspondiente al Word Count clásico

Ya que estos conjuntos de datos tienen, aproximadamente, el mismo tamaño, se atribuye esta diferencia a la disposición de la información. En el caso de *dic*, al ser un texto convencional sin una estandarización previa, la cantidad de líneas de texto es mucho menor que en el caso *word*, donde se ha dispuesto a una palabra por línea. Tras ejecutar los datasets del grupo *dic*, previamente transformados al formato words de una palabra por línea, obtenemos un rendimiento tres veces peor. Lo vemos en las siguientes gráficas:

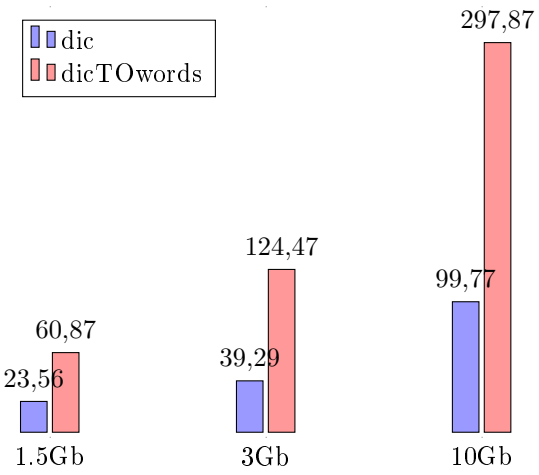


Figura 4.13: Tiempos (en segundos) de ejecución para la aplicación correspondiente al Word Count clásico y para los dataset del grupo dic y su estandarización

Observamos que se da un aumento de los tiempos de, aproximadamente, el 200 % respecto al tiempo original. Ocurre lo mismo para las demás aplicaciones, con un porcentaje de aumento también entorno al 200 %. Esto junto con que se trata de la única diferencia entre los datasets, se hace evidente que, aunque la cantidad de información sea la misma, como se están procesando los datos de modo que cada elemento inicial es una línea del texto, procesar una cantidad mucho más elevada de líneas dispara el tiempo en ejecución. Por tanto, incluir la normalización de los datos en nuestras aplicaciones de Spark es mucho más eficiente. De hecho, el trabajo previo de preparación de los datos, además de llevar tiempo, disminuye la eficiencia de las aplicaciones.

Por otra parte, y al margen de las aplicaciones anteriormente vistas, se van a realizar pruebas de rendimiento respecto a la trasmisión de datos (*broadcast*). Para ello, se han desarrollado dos versiones de una misma aplicación, la cual realiza un *join* entre las frecuencias de las palabras contenidas en el dataset escogido y un pequeño RDD con las letras del abecedario. La idea es cruzar ambos RDDs, haciendo la transmisión del RDD pequeño y sin hacerla. Para aplicar el *broadcast* al RDD hay que procesar el RDD previamente. A continuación, la versión sin trasmisión y, después, con ella:

```
data = sc.textFile(data_file)
words = data.flatMap(cleanData)

vocalsDim = sc.textFile('/user/janira/vocalDim.txt').map(cleanData2).map(lambda x: (x
    [0], x[1]))

frequencies= words.filter(lambda x: x != '').map(lambda x: (x, 1)).reduceByKey(lambda
    a,b: a+b)

startWith = frequencies.map(lambda x: (x[0][0], (x[0],x[1])))

startWithFreqs = frequencies.map(lambda x: (x[0][0], x[1]))

wordsVocalsDim = startWithFreqs.join(vocalsDim)
print(wordsVocalsDim.collect()[0])
```

```
data = sc.textFile(data_file)
words = data.flatMap(cleanData)

vocalsDim = sc.textFile('/user/janira/vocalDim.txt').map(cleanData2).map(lambda x: (x
    [0], x[1]))
vocalsDimBc = sc.broadcast({ vocalsDim.collect()[i][0] : vocalsDim.collect()[i][1] for
     i in range (0,len(vocalsDim.collect())) })

frequencies= words.filter(lambda x: x != '').map(lambda x: (x, 1)).reduceByKey(lambda
    a,b: a+b)

startWith = frequencies.map(lambda x: (x[0][0], (x[0],x[1])))

startWithFreqs = frequencies.map(lambda x: (x[0][0], x[1]))
```

```
12
13    wordsVocalsDim = startWithFreqs.map(lambda x: (x[0], (x[1], vocalsDimBc.value[x[0]])))
14    print(wordsVocalsDim.collect()[0])
```

La trasmisión se tiene que realizar sobre una variable de solo lectura, por lo que es necesario aplicar la función *collect* sobre el RDD de menor tamaño y transformarlo en un diccionario para posibilitar el *join* posterior con el dataset. Se han registrado, para todos los datasets del grupo *dic*, peores tiempos en la versión con *broadcast*. Por ejemplo, para el dataset *dic_100*, con unos 11 segundos de diferencia entre los mejores tiempos de cada versión:

- **app_join_initials:**

	1º	2º	3º
Tiempo	100.841 seg	102.586 seg	102.742 seg
Memoria driver	1500Mb	600Mb	600Mb
Número de ejecutores	2	2	4
Memoria por ejecutor	600Mb	3GB	1500Mb
Núcleos	4	4	4

- **app_join_initials_broadcast:**

	1º	2º	3º
Tiempo	111.979 seg	112.413 seg	113.962 seg
Memoria driver	3GB	600Mb	600Mb
Número de ejecutores	2	2	4
Memoria por ejecutor	600Mb	1500Mb	600Mb
Núcleos	4	4	4

Con estos resultados, se puede concluir que el procesado que se realiza sobre el RDD y su trasmisión, proporcionan disminución de la velocidad en las ejecuciones. Sin embargo, si aplicamos la función *collect* antes de la trasmisión, mantenemos los tiempos respecto a la aplicación original:

```
1     data = sc.textFile(data_file)
2     words = data.flatMap(cleanData)
3
4     vocalsDim = sc.textFile('/user/janira/vocalDim.txt').map(cleanData2).map(lambda x: (x
          [0], x[1]))
5     vocalsDimCollect = vocalsDim.collect()
6     vocalsDimBc = sc.broadcast({ vocalsDimCollect[i][0] : vocalsDimCollect[i][1] for i in
          range (0,len(vocalsDimCollect)) })
7
8     frequencies= words.filter(lambda x: x != '').map(lambda x: (x, 1)).reduceByKey(lambda
          a,b: a+b)
9
10    startWith = frequencies.map(lambda x: (x[0][0], (x[0],x[1])))
11
12    startWithFreqs = frequencies.map(lambda x: (x[0][0], x[1]))
13
14    wordsVocalsDim = startWithFreqs.map(lambda x: (x[0], (x[1], vocalsDimBc.value[x[0]])))
15    print(wordsVocalsDim.collect()[0])
```

Vistos los resultados para los RDDs, se hace evidente que la necesidad de procesar el RDD que se va a transmitir produce un aumento del tiempo de ejecución. Por ello, se han realizado pruebas con la misma estructura pero utilizando DataFrames, con el objetivo de poder sacar conclusiones sobre la utilidad en el uso de la trasmisión de variables. A continuación se muestra el código de la aplicación con *broadcast* utilizada en el caso del trabajo con DataFrames:

```
1     df = spark.read.text(data_file).withColumnRenamed('value','word')
```

```
2
3   vocalsDim = spark.read.csv('/user/janira/vocalDim.txt')
4
5   cleanDf = df.filter(col('word') != '').withColumn('word', regexp_replace(col('word'),'
        [^\sa-zA-Z0-9]', ''))
6
7   frequencies= cleanDf.withColumn('count', lit(1)).groupBy('word').sum('count').
        withColumnRenamed('sum(count)','frequencies')
8
9   startWith = frequencies.withColumn('initial', substring(col('word'),1,1))
10
11  startWithFreqs = startWith.groupBy('initial').sum('frequencies')
12
13  wordsVocalsDim = startWithFreqs.join(broadcast(vocalsDim), startWithFreqs.initial ==
        vocalsDim._c0)
14  wordsVocalsDim.show()
```

Se muestran, en las siguientes gráficas de dispersión, los tiempos obtenidos con la versión sin *broadcast* (*app*) y con la versión con *broadcast* (*broadcast*):

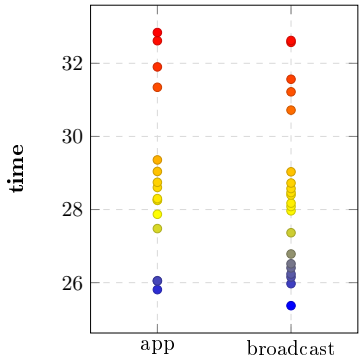


Figura 4.14: Gráfica de dispersión respecto al dataset dic_15, comparando los tiempos de ejecución con y sin *broadcast*

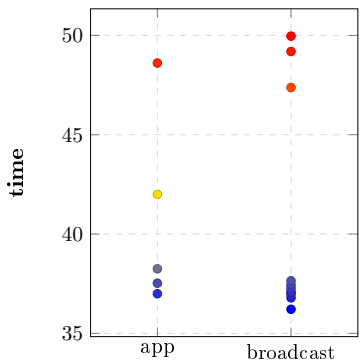


Figura 4.15: Gráfica de dispersión respecto al dataset dic_30, comparando los tiempos de ejecución con y sin *broadcast*

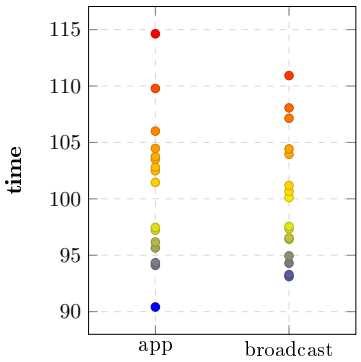


Figura 4.16: Gráfica de dispersión respecto al dataset dic_100, comparando los tiempos de ejecución con y sin *broadcast*

En los dos primeros casos, se ve una leve mejora con la versión *broadcast* para los mejores tiempos. Sin embargo, en el caso del dataset más grande, no parece beneficiar esta estrategia.

4.7. Configuración del clúster

La asignación de los recursos disponibles del clúster sobre la ejecución de nuestra aplicación juega un papel clave en la optimización de tiempo y, por supuesto, en la prevención de fallos por falta de memoria. En el vídeo *Top 5 Mistakes When Writing Spark Applications (Grover, M. & Malaska, T., 16 junio 2016, 1m11s)* se expone la correcta asignación para un clúster de 6 nodos con 16 núcleos y 64 GB de memoria cada uno. La idea base para el ajuste de recursos será que un ejecutor no tenga más de 5 núcleos. Con esta premisa y teniendo en cuenta la reserva de recursos para el *Application Master* (1 ejecutor, 1 núcleo y 1 GB de memoria) se establece:

- Se dispone de 15 x 6 = 90 núcleos para repartir en los ejecutores. Como se quieren 5 núcleos por ejecutor: 90/5 = 18 ejecutores. Restando el reservado para el AM: 17 ejecutores.
- Hay 6 nodos, por lo que se dispondrá de 3 ejecutores por nodo.
- 64GB por nodo, menos el reservado para el AM: 63GB por nodo. Deberíamos tener 63/3 = 21GB por ejecutor, pero se recomienda liberar un 0.7 % de la memoria. Así, se tendrán alrededor de 19GB asignables a la memoria de cada ejecutor.

Esta asignación corresponde con una buena práctica a la hora de repartir los recursos, pero no es la asignación que ha de tomarse en todos los casos. Obviamente, dependerá de las características de nuestro clúster y la vía para dar con el mejor ajuste será el ensayo y error. Así, en este apartado se ha realizado el mayor número de ejecuciones, para poder ajustar nuestra elección en cuanto al número de recursos que se deben asignar a nuestras aplicaciones, teniendo en cuenta el tipo y el tamaño del dataset con el que estamos trabajando.

Se han realizado alrededor de 1000 ejecuciones por dataset, abarcando un rango de configuraciones que resultara interesante. Tras ir ajustando las variables para que resultaran significativas, las ejecuciones finales para las que se han tenido en cuenta los resultados han constado de configuraciones resultado de combinar:

- Entre 600 Mb y 6GB para la memoria del driver
- 1 o 2 ejecutores por nodo.
- Entre 600 Mb y 6GB de memoria por ejecutor

- 1, 2, 3 y 4 núcleos por ejecutor.

Las pruebas se centrarán en los máximos y mínimos posibles respecto a la memoria asignada y en las configuraciones con 1 o 2 ejecutores por nodo debido al tamaño del clúster. Se tienen 6 nodos con 4 núcleos cada uno. Por lo tanto, las posibilidades de configuración, teniendo en cuenta que el máximo de núcleos por nodo es 4, son:

- 1 ejecutor con 1, 2, 3 o 4 núcleos.

- 2 ejecutores con 1 o 2 núcleos por ejecutor.

Si se estableciera un número mayor de ejecutores, a partir de los dos núcleos por ejecutor, en realidad se estará estableciendo un solo ejecutor. Al no encontrar suficientes recursos, Spark asignará la cantidad disponible, en este caso uno o dos ejecutores (según el número de núcleos).

4.7.1. Resultados

En los resultados obtenidos destaca principalmente la diferenciación de tiempos según los núcleos establecidos. Respecto a esta variable, por dataset, el comportamiento de las tres aplicaciones es el mismo:

a) **dic_15 vs words_15**

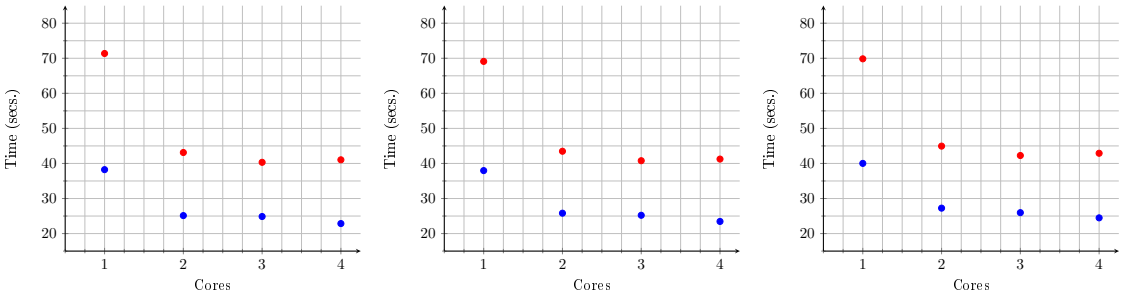


Figura 4.17: De izquierda a derecha: Word Count, Word Count Level 2 y Word Count Plus. Resultados obtenidos para los datasets dic_15 (en azul) y words_15 (en rojo).

El número de núcleos que tiene los mejores tiempos en este caso es el máximo, 4. Aunque las diferencias no son muy distantes, en los resultados se observa una clara diferenciación entre los tiempos resultantes para 4 núcleos y los tiempos para 2 y 3, lo cuales son muy semejantes. El caso de un solo núcleo se refleja a modo informativo ya que no tiene interés debido a la falta de paralelismo en el nodo de trabajo y, en consecuencia, sus malos resultados en esta comparativa.

b) **dic_30 vs words_30**

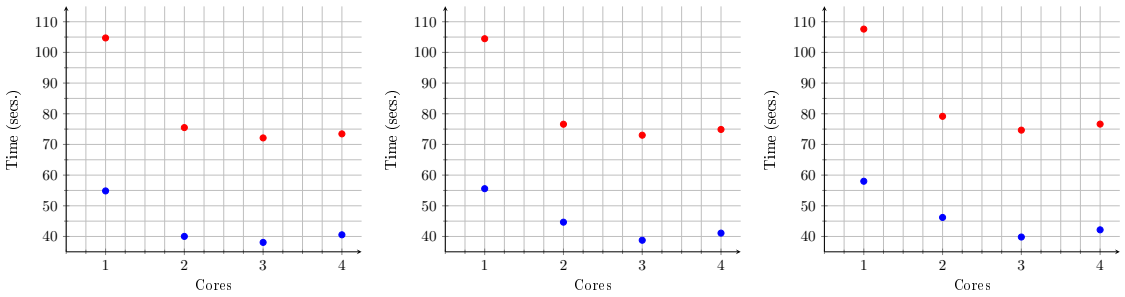


Figura 4.18: De izquierda a derecha: Word Count, Word Count Level 2 y Word Count Plus. Resultados obtenidos para los datasets dic_30 (en azul) y words_30 (en rojo).

En este caso, los mejores tiempos corresponden a 3 núcleos en ambos datasets. Se podría determinar el aumento de tamaño de los datos ha provocado que el número de ejecutores óptimo varíe. Además, la diferencia de tiempos entre el segundo y tercer puesto (4 y 2 núcleos) es mayor que para el caso anterior (3 y 2 núcleos).

c) **dic_100 vs words_100**

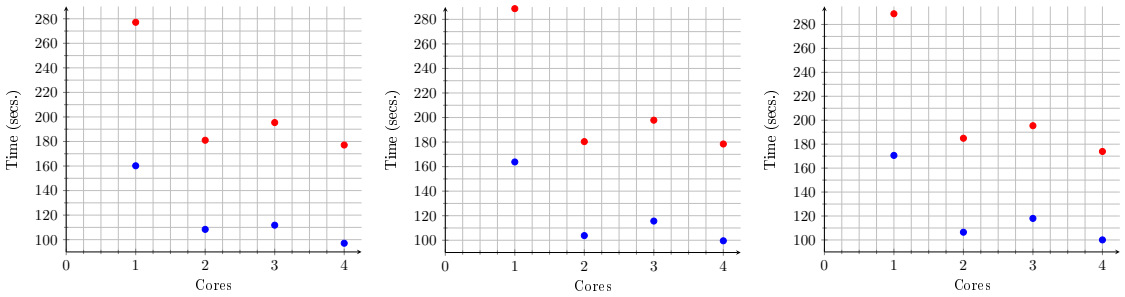


Figura 4.19: De izquierda a derecha: Word Count, Word Count Level 2 y Word Count Plus. Resultados obtenidos para los datasets dic_100 (en azul) y words_100 (en rojo).

Para los datasets de mayor tamaño observamos que 4 núcleos es la mejor opción, seguida de 2 núcleos.

Debido a los resultados anteriores, en adelante se centrarán las ejecuciones en los datasets del grupo *dic*. Además, se podrá observar que los resultados correspondientes a dos ejecutores y dos núcleos no estarán presentes en los mejores tiempos registrados. Por tanto, se puede concluir que en los casos que estamos tratando es preferible tener un ejecutor por nodo con tres o cuatro núcleos, que dos ejecutores con dos núcleos cada uno.

Se podría determinar que la diferencia, en el caso de los datasets de unos 3GB de tamaño, se debe al resto de parámetros de configuración, pero a la hora de observar dichas variables en sus tiempos ganadores, no se puede determinar una configuración que destaque. Existe disparidad entre la memoria asignada al *driver* y la memoria asignada a los ejecutores. Aquí se muestran la configuración de los tres cuatro mejores tiempos para cada una de las aplicaciones:

- **dic_30**

1. **Word Count:**

Tabla 4.1: Mejores resultados para el dataset dic_30 y la aplicación Word Count.

	1º	2º	3º	4º
Tiempo	38.068 seg	38.114 seg	38.161 seg	38.191 seg
Memoria driver	1500Mb	3GB	3GB	3GB
Número de ejecutores	1	1	1	1
Memoria por ejecutor	1500Mb	1500Mb	3GB	6GB
Núcleos	3	3	3	3

2. **Word Count Level 2:**

Tabla 4.2: Mejores resultados para el dataset dic_30 y la aplicación Word Count Level 2.

	1º	2º	3º	4º
Tiempo	38.772 seg	38.797 seg	38.856 seg	38.998 seg
Memoria driver	3GB	600Mb	1500Mb	3GB
Número de ejecutores	1	1	1	1
Memoria por ejecutor	3GB	3GB	1500Mb	6GB
Núcleos	3	3	3	3

3. **Word Count Plus:**

Tabla 4.3: Mejores resultados para el dataset dic_30 y la aplicación Word Count Plus.

	1º	2º	3º	4º
Tiempo	39.818 seg	39.936 seg	39.944 seg	40.046
Memoria driver	600Mb	600Mb	3GB	3GB
Número de ejecutores	1	1	1	1
Memoria por ejecutor	1500Mb	3GB	600Mb	3GB
Núcleos	3	3	3	3

De hecho, esta falta de conclusiones en la elección de la memoria se da en el resto de datasets y aplicaciones:

- **dic_15**

1. **Word Count:**

Tabla 4.4: Mejores resultados para el dataset dic_15 y la aplicación Word Count.

	1º	2º	3º	4º
Tiempo	22.854 seg	22.947 seg	22.958 seg	22.992 seg
Memoria driver	3GB	3GB	3GB	3GB
Número de ejecutores	1	1	1	1
Memoria por ejecutor	6GB	3GB	600Mb	600Mb
Núcleos	4	4	4	4

2. **Word Count Level 2:**

Tabla 4.5: Mejores resultados para el dataset dic_15 y la aplicación Word Count Level 2.

	1º	2º	3º	4º
Tiempo	23.457 seg	23.465 seg	23.605 seg	23.627 seg
Memoria driver	3GB	3GB	1500Mb	600Mb
Número de ejecutores	1	1	1	1
Memoria por ejecutor	1500Mb	6GB	6GB	6GB
Núcleos	4	4	4	4

3. **Word Count Plus:**

Tabla 4.6: Mejores resultados para el dataset dic_15 y la aplicación Word Count Plus.

	1º	2º	3º	4º
Tiempo	24.484 seg	24.795 seg	24.833 seg	24.853 seg
Memoria driver	1500Mb	600Mb	3GB	1500Mb
Número de ejecutores	1	1	1	1
Memoria por ejecutor	1500Mb	6GB	6GB	3GB
Núcleos	4	4	4	4

- **dic_100**

1. **Word Count:**

Tabla 4.7: Mejores resultados para el dataset dic_100 y la aplicación Word Count.

	1º	2º	3º	4º
Tiempo	97.081 seg	99.496 seg	99.606 seg	99.617 seg
Memoria driver	1500Mb	600Mb	600Mb	1500Mb
Número de ejecutores	1	1	1	1
Memoria por ejecutor	3GB	3GB	1500Mb	3GB
Núcleos	4	4	4	4

2. **Word Count Level 2:**

Tabla 4.8: Mejores resultados para el dataset dic_100 y la aplicación Word Count Level 2.

	1º	2º	3º	4º
Tiempo	99.544 seg	99.568 seg	99.847 seg	100.645 seg
Memoria driver	600Mb	3GB	3GB	600Mb
Número de ejecutores	1	1	1	1
Memoria por ejecutor	3GB	3GB	1500Mb	6GB
Núcleos	4	4	4	4

3. **Word Count Plus:**

Tabla 4.9: Mejores resultados para el dataset dic_100 y la aplicación Word Count Plus.

	1º	2º	3º	4º
Tiempo	100.05 seg	100.679 seg	101.219 seg	101.415 seg
Memoria driver	600Mb	1500Mb	1500Mb	600Mb
Número de ejecutores	1	1	1	1
Memoria por ejecutor	1500Mb	600Mb	1500Mb	1500Mb
Núcleos	4	4	4	4

4.8. Particionado

Una partición representa una unidad en la ejecución paralela y corresponde con una tarea. Tener muy pocas particiones puede hacer que no se aproveche al máximo las ventajas del paralelismo al perder concurrencia. Tener demasiadas puede provocar tiempos excesivos en la programación de las tareas. En general, se recomienda tener, al menos, el doble de particiones que de núcleos disponibles para la ejecución.

Se llama particionador al objeto que determina cómo están particionados por clave los elementos de pares (clave, valor) en un RDD. Spark es compatible con dos tipos de particionados: *HashPartitioner* y *RangePartitioner*, aunque, como se verá más adelante, se pueden crear particionadores personalizados.

HashPartitioner

Particionador por defecto de los RDDs de pares. Determina la clasificación de la partición derivada basándose en el valor *hash* de la clave y el número de particiones dadas por parámetro:

$$p = \text{key.hashCode()} \bmod (\text{numPartitions})$$

Esta función *hash* crea tantos contenedores como número de particiones y clasificará las claves basándose en la operación módulo, es decir, en el resto que se obtiene de la división entre las claves y *numPartitions*. Si no se especifica el parámetro *numPartitions*, Spark establece el número de particiones por defecto. Si este parámetro de configuración tampoco estuviera definido, Spark toma el mayor número de particiones que se haya dado dentro del linaje de la partición con la que se está trabajando.

RangePartitioner

Este particionador utiliza una división de rango para la clasificación de las claves del RDD en distintas particiones. Se tendrá que dar como argumentos tanto el número de particiones resultantes deseadas como el RDD sobre el que se quiere actuar. Se determinan límites de rango para cada partición mediante muestreo, asegurando una distribución igualitaria de los registros entre las particiones. Cada registro se incluirá en la partición cuyo rango incluya la clave del par.

Se tienen algunos métodos para modificar el esquema de partición según el tipo de RDDs con los que se esté trabajando. Para RDDs que contienen un tipo de objeto genérico se utilizan las funciones *coalesce()* y *repartition()* en las cuales se establece por parámetro el número de particiones que tendrán los datos. *coalesce()* no requiere de barajado de datos entre nodos (a diferencia de *repartition*) puesto que, si el número de particiones que se quieren establecer es menor que el presente, se realiza un combinado de particiones para recudir su número. En caso de querer un número mayor de particiones del que hay, esta función tiene el mismo comportamiento que *repartition()*, el cual genera una partición *hash*.

partitionBy()

Para RDDs de pares existe la operación *partitionBy()*. Esta función permite aplicar un método de partición personalizada que se proporciona por parámetro y realiza el *shuffle* con el nuevo modelo de partición. A diferencia de las funciones anteriores, proporciona un particionado conocido y, en consecuencia, mayor control sobre la localización de los datos. Esto puede evitar el *shuffle* aun teniendo extensas dependencias entre las particiones.

- Utilizando *HashPartitioner*: se pasa por parámetro el número de particiones y usa el particionador *hash* por defecto.

```
rdd = sc.parallelize([1,2,3,4,5]).map(lambda x: (x,x))
.partitionBy(2)

#rdd.glom().\textit{collect()}-> [[1,3,5], [2,4]]
```

- Creando una función de partición: en este ejemplo, en vez de utilizar el particionador *hash*, definimos una función que devuelve el código ASCII de un carácter (ord(a)=97, ord(b)=98, ord(c)=99, ord(d)=100 y ord(e)=101). Se pasa esta función como segundo parámetro y se realizará el cálculo: ord(x) % 3 para asignar cada elemento a una de las 3 particiones.

```
def ascii\_partitioner(x):
return ord(x)

rdd = sc.parallelize(['a','b','c','d','e']).map(lambda x: (x,1))
.partitionBy(3, ascii\_partitioner)

#rdd.glom().\textit{collect()}-> [['c'], ['a','d'], ['b','e']]
```

Es importante tener en cuenta que las trasformaciones *wide* modifican la partición de un RDD. Aunque siempre se puede determinar el número de particiones que se quiere establecer para el RDD resultante en una transformación *wide*, si realizamos varias transformaciones, entender cómo se ha particionado el RDD es una información muy útil en situaciones en las que es posible evitar el *shuffle*. En cuanto a las transformaciones *narrow*, existen algunas que preservan el particionado. Aplicar una transformación que únicamente afecte a los valores de los pares clave/valor no garantiza que el RDD resultante haya conservado el particionado original. Por ello, si se pretende utilizar *map* o *flatMap* sin hacer modificaciones en la clave la mejor opción es utilizar *mapValues* ya que mantiene las claves y el particionado. También tiene esta consistencia las funciones *mapPartitions*, *filter* si la opción *preservesPartititoning* está establecido en *true*. Otra cuestión que tiene gran importancia en este sentido es la ubicación compartida de los RDDs unida a la presencia del mismo particionador en ellos. Cuando se dan ambas situaciones se evita el *shuffle* en operaciones que requieren el orden aleatorio para la agrupación como *cogroup* y *join*.

4.8.1. Resultados

Se ha continuado ejecutando con las diferentes configuraciones (exceptuando el caso de 1 núcleo por ejecutor) con la intención de encontrar alguna relación de eficiencia en tiempo y memoria asignada.

En las ejecuciones del apartado anterior, al no haber modificado el número de particiones de nuestros RDDs, podemos ver el número de particiones que la lógica Spark establece. Teniendo un nivel predeterminado de paralelismo definido en *SparkContext* de 24, obtenemos las siguientes particiones para todos los RDDs en las tres aplicaciones:

- **dic_15:** 25 particiones
- **dic_30:** 49 particiones
- **dic_100:** 161 particiones

El objetivo de este apartado es jugar con el número de particiones en los distintos RDDs para ver si estas modificaciones afectan significativamente el tiempo de ejecución de las aplicaciones. Para ello, se ha añadido una variable extra (que se identificará en el código como la variable *internal_param[7]*) en las ejecuciones del apartado anterior, que será el parámetro de multiplicación del valor:

(número de ejecutores) * (número de núcleos por ejecutor)

y cuyo resultado establecerá el número de particiones en las que se reparticionarán los RDDs.
A continuación, se van a ver los resultados por dataset, ya que el número de particiones que Spark establece por defecto es diferente según el tamaño del dataset a procesar.

1. **dic_15**

   a) **word_count:** mejora aumentando las particiones del RDD que resulta de la lectura del dataset mediante la función *textFile*, en la cual establecemos en el segundo parámetro el número de particiones mínimas que va a tener el RDD:

```
data = sc.textFile(data_file, int(int(sc.getConf().get("spark.executor.
    cores")) * int(sc.getConf().get("spark.executor.instances")) * float
    (internal_param[7])))

words = data.flatMap(cleanData)
frequencies= words.filter(lambda x: x != '').map(lambda x: (x, 1)).
    reduceByKey(lambda a,b: a+b)
print('Example of words frequencies:', frequencies.collect()[1])
```

   Los tiempos ganadores corresponden con la modificación de las 25 particiones que establece Spark por defecto a 40 y 56 particiones:

	1º	2º	3º
Tiempo	21.767 seg	22.019 seg	22.107 seg
Memoria driver	600Mb	1500Mb	3GB
Número de ejecutores	1	1	1
Memoria por ejecutor	1500Mb	1500Mb	1500Mb
Núcleos	4	4	4
Factor de multiplicación	10	14	14
Nivel de paralelismo	40	56	56

   Como peores tiempos para esta versión de la aplicación tenemos:

	1º	2º	3º
Tiempo	27.193 seg	25.887 seg	25.478 seg
Memoria driver	3GB	1500Mb	3GB
Número de ejecutores	1	1	1
Memoria por ejecutor	1500Mb	1500Mb	600Mb
Núcleos	4	4	4
Factor de multiplicación	10	10	10
Nivel de paralelismo	40	40	40

   Para un número de particiones mucho más pequeño como 2, 4 u 8 particiones, encontramos valores entre los diez mejores tiempos y los diez peores

   b) **word_count_level_2:** mejora reduciendo las particiones del RDD que resulta de la suma de las apariciones de cada palabra:

```
data = sc.textFile(data_file)

words = data.flatMap(cleanData)
frequencies= words.filter(lambda x: x != '').map(lambda x: (x, 1)).
    reduceByKey(lambda a,b: a+b).\textsl{repartition}(int(int(sc.getConf
    ().get("spark.executor.cores")) * int(sc.getConf().get("spark.
    executor.instances")) * float(internal_param[7])))

initialsFreqs = frequencies.map(lambda x: (x[0][0], (x[0], x[1])))
groupInitialFreqs = initialsFreqs.groupByKey()
print('Example of words frequencies group by initial:',
    groupInitialFreqs.collect()[1])
```

Los tiempos ganadores corresponden con la disminución de las 25 particiones que establece Spark por defecto a 4 y 6 particiones:

	1º	2º	3º
Tiempo	23.1 seg	23.181 seg	23.19 seg
Memoria driver	1500Mb	3GB	3GB
Número de ejecutores	1	1	1
Memoria por ejecutor	1500Mb	3GB	1500Mb
Núcleos	4	4	4
Factor de multiplicación	1	1.5	1
Nivel de paralelismo	2	6	4

Como peores tiempos para esta versión de la aplicación tenemos:

	1º	2º	3º
Tiempo	30.713 seg	30.542 seg	27.837 seg
Memoria driver	600Mb	3GB	600Mb
Número de ejecutores	1	1	1
Memoria por ejecutor	600Mb	1500Mb	600Mb
Núcleos	4	4	4
Factor de multiplicación	1.5	0.5	0.5
Nivel de paralelismo	6	2	2

c) **word_count_plus:** mejora reduciendo las particiones del RDD que resulta de la suma de las apariciones de cada palabra:

```
1        data = sc.textFile(data_file)
2
3        words = data.flatMap(cleanData)
4        frequencies= words.filter(lambda x: x != '').map(lambda x: (x, 1)).
             reduceByKey(lambda a,b: a+b).\textsl{repartition}(int(int(sc.getConf
             ().get("spark.executor.cores")) * int(sc.getConf().get("spark.
             executor.instances")) * float(internal_param[7])))
5
6        initialsFreqs = frequencies.map(lambda x: (x[0][0], (x[0], x[1])))
7        groupInitialFreqs = initialsFreqs.groupByKey()
8
9        reduceInitialFreqs = initialsFreqs.map(lambda x: (x[0], x[1][1]))
10
11       totalInfoFreqs = reduceInitialFreqs.join(initialsFreqs)
12       print('Example of result: ', totalInfoFreqs.collect()[1])
```

Los tiempos ganadores corresponden con la disminución de las 25 particiones que establece Spark por defecto a 2 y 4 particiones:

	1º	2º	3º
Tiempo	23.404 seg	23.444 seg	23.723 seg
Memoria driver	3GB	600Mb	600Mb
Número de ejecutores	1	1	1
Memoria por ejecutor	3GB	3GB	600Mb
Núcleos	4	4	4
Factor de multiplicación	0.5	0.5	1
Nivel de paralelismo	2	2	4

Como peores tiempos para esta versión de la aplicación tenemos:

	1º	2º	3º
Tiempo	27.606 seg	27.283 seg	27.076 seg
Memoria driver	600Mb	600Mb	3GB
Número de ejecutores	1	1	1
Memoria por ejecutor	600Mb	3GB	3GB
Núcleos	4	4	4
Factor de multiplicación	10	10	10
Nivel de paralelismo	40	40	40

2. **dic_30**

a) **word_count:** encontramos sólo dos casos de mejora, para el mismo caso de aplicación que para **dic_15**:

```
data = sc.textFile(data_file, int(int(sc.getConf().get("spark.executor.
    cores")) * int(sc.getConf().get("spark.executor.instances")) * float(
    internal_param[7])))

words = data.flatMap(cleanData)
frequencies= words.filter(lambda x: x != '').map(lambda x: (x, 1)).
    reduceByKey(lambda a,b: a+b)
print('Example of words frequencies:', frequencies.collect()[1])
```

El tiempo ganador corresponde con el aumento de las 49 particiones que establece Spark por defecto a 56 particiones:

	1º	2º
Tiempo	37.758 seg	37.907 seg
Memoria driver	600Mb	3GB
Número de ejecutores	1	1
Memoria por ejecutor	3GB	6GB
Núcleos	4	3
Factor de multiplicación	12	5
Nivel de paralelismo	56	49

Se observa en el caso del segundo tiempo que en vez de las 15 particiones que deberían establecerse, Spark determina 49 particiones, como en los casos iniciales. Esto se debe a que la función *textFile()* tiene como segundo argumento *minPartitions*, que establece el número mínimo que Spark considera para los datos en función del tamaño. Este valor se puede modificar pero siempre a la alta. Al introducir un número menor de particiones que el mínimo, Spark dará el valor *minPartitions*. Llama la atención que el mejor tiempo conseguido haya sido con 4 núcleos por ejecutor, ya que hasta ahora los tiempos más pequeños se obtenían con 3 núcleos.

Como peores tiempos para esta versión de la aplicación tenemos:

	1º	2º	3º
Tiempo	49.319 seg	49.244 seg	49.218 seg
Memoria driver	600Mb	600Mb	600Mb
Número de ejecutores	1	1	1
Memoria por ejecutor	3GB	600Mb	3GB
Núcleos	3	3	3
Factor de multiplicación	20	20	40
Nivel de paralelismo	60	60	120

b) **word_count_level_2:** en este caso, la versión de la aplicación que consiguen mejorar el tiempo de ejecución es la que reduce las particiones del RDD que resulta de la suma de las apariciones de cada palabra:

```
data = sc.textFile(data_file)

words = data.flatMap(cleanData)
frequencies= words.filter(lambda x: x != '').map(lambda x: (x, 1)).
    reduceByKey(lambda a,b: a+b).\textsl{repartition}(int(int(sc.getConf
    ().get("spark.executor.cores")) * int(sc.getConf().get("spark.
    executor.instances")) * float(internal_param[7])))

initialsFreqs = frequencies.map(lambda x: (x[0][0], (x[0], x[1])))
groupInitialFreqs = initialsFreqs.groupByKey()
print('Example of words frequencies group by initial:',
    groupInitialFreqs.collect()[1])
```

Los tiempos ganadores corresponden con la disminución de las 161 particiones que establece Spark por defecto a 1, 6 y 15 particiones:

	1º	2º	3º
Tiempo	39.225 seg	39.294 seg	39.313 seg
Memoria driver	1500Mb	3GB	3GB
Número de ejecutores	1	1	1
Memoria por ejecutor	3GB	3GB	1500Mb
Núcleos	3	3	3
Factor de multiplicación	0.5	2	5
Nivel de paralelismo	1	6	15

Como peores tiempos para esta versión de la aplicación tenemos:

	1º	2º	3º
Tiempo	41.326 seg	40.868 seg	40.75 seg
Memoria driver	1500Mb	3GB	1500Mb
Número de ejecutores	1	1	1
Memoria por ejecutor	1500Mb	3GB	1500Mb
Núcleos	3	3	3
Factor de multiplicación	20	4	12
Nivel de paralelismo	60	12	36

c) **word_count_plus:** en este caso encontramos una ligera mejora reduciendo las particiones del RDD *frequencies* y de los que, en la definición del último RDD, se cruzarán mediante la función *join()*:

```
data = sc.textFile(data_file)

words = data.flatMap(cleanData)
frequencies= words.filter(lambda x: x != '').map(lambda x: (x, 1)).
    reduceByKey(lambda a,b: a + b).reduceByKey(lambda a,b: a+b).\textsl{
    repartition}(int(int(sc.getConf().get("spark.executor.cores")) * int(
    sc.getConf().get("spark.executor.instances")) * float(internal_param
    [7])))

initialsFreqs = frequencies.map(lambda x: (x[0][0], (x[0], x[1]))).
    reduceByKey(lambda a,b: a + b).\textsl{repartition}(int(int(sc.getConf
    ().get("spark.executor.cores")) * int(sc.getConf().get("spark.executor
    .instances")) * float(internal_param[7])))

reduceInitialFreqs = initialsFreqs.map(lambda x: (x[0], x[1][1])).
    reduceByKey(lambda a,b: a + b).\textsl{repartition}(int(int(sc.getConf
    ().get("spark.executor.cores")) * int(sc.getConf().get("spark.executor
    .instances")) * float(internal_param[7])))

totalInfoFreqs = reduceInitialFreqs.join(initialsFreqs)
print('Example of result: ', totalInfoFreqs.collect()[1])
```

Los tiempos ganadores corresponden con la disminución de las 49 particiones que establece Spark por defecto a 1 y 6 particiones:

	1º	2º	3º
Tiempo	39.328 seg	39.539 seg	39.546 seg
Memoria driver	600Mb	1500Mb	600Mb
Número de ejecutores	1	1	1
Memoria por ejecutor	6GB	1500Mb	1500Mb
Núcleos	3	3	3
Factor de multiplicación	0.5	2	2
Nivel de paralelismo	1	6	6

Como peores tiempos para esta versión de la aplicación tenemos:

	1º	2º	3º
Tiempo	48.716 seg	47.717 seg	47.091 seg
Memoria driver	3GB	1500Mb	600Mb
Número de ejecutores	1	1	1
Memoria por ejecutor	3GB	1500Mb	1500Mb
Núcleos	3	3	3
Factor de multiplicación	40	40	40
Nivel de paralelismo	120	120	120

3. **dic_100**

a) **word_count:** para esta aplicación no encontramos resultados que mejoren el tiempo de ejecución con la repartición de ninguno de los RDDs. Los resultados son muy similares a los del apartado de configuración, sin llegar a reducirlos.

b) **word_count_level_2:** se observan mejoras para la misma aplicación que en el caso de *dic_30*. Los tiempos ganadores corresponden con la disminución de las 161 particiones que establece Spark por defecto a 6, 2 y 20 particiones:

	1º	2º	3º
Tiempo	98.58 seg	98.974 seg	99.084 seg
Memoria driver	3GB	3GB	1500Mb
Número de ejecutores	1	1	1
Memoria por ejecutor	1500Mb	1500Mb	1500Mb
Núcleos	4	4	4
Factor de multiplicación	1.5	0.5	5
Nivel de paralelismo	6	2	20

Como peores tiempos para esta versión de la aplicación tenemos:

	1º	2º	3º
Tiempo	121.518 seg	114.401 seg	113.534 seg
Memoria driver	1500Mb	3GB	3GB
Número de ejecutores	1	1	1
Memoria por ejecutor	3GB	600Mb	3GB
Núcleos	4	4	4
Factor de multiplicación	0.5	2	2
Nivel de paralelismo	2	8	8

c) **word_count_plus:** se obtiene mejora reduciendo las particiones del RDD *frequencies* y de los RDD que posteriormente se cruzarán mediante la función *join()*:

```
1       data = sc.textFile(data_file)
2
```

```
3       words = data.flatMap(cleanData)
4       frequencies= words.filter(lambda x: x != '').map(lambda x: (x, 1)).
            reduceByKey(lambda a,b: a + b).reduceByKey(lambda a,b: a+b).\textsl{
            repartition}(int(int(sc.getConf().get("spark.executor.cores")) * int(
            sc.getConf().get("spark.executor.instances")) * float(internal_param
            [7])))
5
6       initialsFreqs = frequencies.map(lambda x: (x[0][0], (x[0], x[1]))).
            reduceByKey(lambda a,b: a + b).\textsl{repartition}(int(int(sc.getConf
            ().get("spark.executor.cores")) * int(sc.getConf().get("spark.executor
            .instances")) * float(internal_param[7])))
7
8       reduceInitialFreqs = initialsFreqs.map(lambda x: (x[0], x[1][1])).
            reduceByKey(lambda a,b: a + b).\textsl{repartition}(int(int(sc.getConf
            ().get("spark.executor.cores")) * int(sc.getConf().get("spark.executor
            .instances")) * float(internal_param[7])))
9
10      totalInfoFreqs = reduceInitialFreqs.join(initialsFreqs)
11      print('Example of result: ', totalInfoFreqs.collect()[1])
```

Los tiempos ganadores corresponden con la disminución de las 161 particiones que establece Spark por defecto a 10, 2 y 80 particiones:

	1º	2º	3º
Tiempo	99.296 seg	99.544 seg	99.779 seg
Memoria driver	1500Mb	1500Mb	1500Mb
Número de ejecutores	1	1	1
Memoria por ejecutor	600Mb	1500Mb	600Mb
Núcleos	4	4	4
Factor de multiplicación	2.5	0.5	20
Nivel de paralelismo	10	2	80

Como peores tiempos para esta versión de la aplicación tenemos:

	1º	2º	3º
Tiempo	129.897 seg	117.411 seg	113.534 seg
Memoria driver	3GB	1500Mb	1500Mb
Número de ejecutores	1	1	1
Memoria por ejecutor	6GB	3GB	6GB
Núcleos	4	4	4
Factor de multiplicación	40	40	20
Nivel de paralelismo	160	160	80

Para el caso de las aplicaciones con más de una acción (*words*), también se han registrado mejoras en los tiempos. Recordemos que este grupo de aplicaciones solo afecta a *Word Count Level 2* y *Word Count Plus*.

1. **dic_15**

   a) **word_count_level_2_jobs:** reduciendo las particiones del RDD que resulta de la suma de las apariciones de cada palabra:

```
1       data = sc.textFile(data_file)
2
3         words = data.flatMap(cleanData)
4         frequencies= words.filter(lambda x: x != '').map(lambda x: (x, 1)).
              reduceByKey(lambda a,b: a+b).repartition(int(int(sc.getConf().get(
              "spark.executor.cores")) * int(sc.getConf().get("spark.executor.
              instances")) * float(internal_param[7])))
5         print('Example of words frequencies:', frequencies.collect()[1])
6
7         initialsFreqs = frequencies.map(lambda x: (x[0][0], (x[0], x[1])))
8         groupInitialFreqs = initialsFreqs.groupByKey()
9         print('Example of words frequencies group by initial:',
              groupInitialFreqs.collect()[1])
```

Se obtiene una ligera mejora de los mejores tiempos:

	1º	2º	3º
Tiempo original	23.831 seg	23.923 seg	24.095 seg
Tiempo	23.513 seg	23.694 seg	23.703 seg
Memoria driver	600Mb	1500Mb	3GB
Número de ejecutores	1	1	1
Memoria por ejecutor	600Mb	600Mb	1500Mb
Núcleos	4	4	4
Factor de multiplicación	0.5	2.5	0.5
Nivel de paralelismo	2	6	2

b) **word_count_plus_jobs:** se obtienen mejoras realizando el reparticionado el RDD *frequencies* y los dos RDDs involucrados en la operación *join()*:

```
1       data = sc.textFile(data_file)
2
3         words = data.flatMap(cleanData)
4         frequencies= words.filter(lambda x: x != '').map(lambda x: (x, 1)).
              reduceByKey(lambda a,b: a+b).repartition(int(int(sc.getConf().get("
              spark.executor.cores")) * int(sc.getConf().get("spark.executor.
              instances")) * float(internal_param[7])))
5         print('Example of words frequencies:', frequencies.collect()[1])
6
7         initialsFreqs = frequencies.map(lambda x: (x[0][0], (x[0], x[1]))).
              repartition(int(int(sc.getConf().get("spark.executor.cores")) * int(
              sc.getConf().get("spark.executor.instances")) * float(internal_param
              [7])))
8         groupInitialFreqs = initialsFreqs.groupByKey()
9         print('Example of words frequencies group by initial:',
              groupInitialFreqs.collect()[1])
10
11        reduceInitialFreqs = initialsFreqs.map(lambda x: (x[0], x[1][1])).
              reduceByKey(lambda a,b: a + b).repartition(int(int(sc.getConf().get(
              "spark.executor.cores")) * int(sc.getConf().get("spark.executor.
              instances")) * float(internal_param[7])))
12        totalInfoFreqs = reduceInitialFreqs.join(initialsFreqs)
13        print('Example of result: ', totalInfoFreqs.collect()[1])
```

Con los siguientes resultados:

	1º	2º	3º
Tiempo original	25.882 seg	26.012 seg	26.079 seg
Tiempo	24.797 seg	25.209 seg	25.324 seg
Memoria driver	1500Mb	1500Mb	1500Mb
Número de ejecutores	1	1	1
Memoria por ejecutor	3GB	1500Mb	3GB
Núcleos	4	4	4
Factor de multiplicación	0.5	0.5	2
Nivel de paralelismo	2	2	8

En ambas aplicaciones los peores tiempos se dan en las versiones de la aplicación con mayor número de particiones. Cuantas más particiones, peor tiempo se obtiene.

2. **dic_30**

a) **word_count_level_2_jobs:** como en el caso de dic_15, se obtiene mejora reduciendo las particiones del RDD que resulta de la suma de las apariciones de cada palabra:

	1º	2º	3º
Tiempo original	39.474 seg	39.679 seg	39.746 seg
Tiempo	38.803 seg	39.189 seg	39.216 seg
Memoria driver	600Mb	600Mb	1500Mb
Número de ejecutores	1	1	1
Memoria por ejecutor	1500Mb	600Mb	3GB
Núcleos	3	3	3
Factor de multiplicación	2	1	1
Nivel de paralelismo	6	3	3

b) **word_count_plus_jobs:** se obtienen mejoras realizando el reparticionado en los dos RDDs involucrados en la operación *join()*:

```
    data = sc.textFile(data_file)

    words = data.flatMap(cleanData)
    frequencies= words.filter(lambda x: x != '').map(lambda x: (x, 1)).
        reduceByKey(lambda a,b: a + b)
    print('Example of words frequencies:', frequencies.collect()[1])

    initialsFreqs = frequencies.map(lambda x: (x[0][0], (x[0], x[1]))).
        reduceByKey(lambda a,b: a + b).repartition(int(int(sc.getConf().get("
        spark.executor.cores")) * int(sc.getConf().get("spark.executor.
        instances")) * float(internal_param[7])))
    groupInitialFreqs = initialsFreqs.groupByKey()
    print('Example of words frequencies group by initial:', groupInitialFreqs.
        collect()[1])

    reduceInitialFreqs = initialsFreqs.map(lambda x: (x[0], x[1][1])).
        reduceByKey(lambda a,b: a + b).repartition(int(int(sc.getConf().get("
        spark.executor.cores")) * int(sc.getConf().get("spark.executor.
        instances")) * float(internal_param[7])))

    totalInfoFreqs = reduceInitialFreqs.join(initialsFreqs)
    print('Example of result: ', totalInfoFreqs.collect()[1])
```

Con los siguientes resultados:

	1º	2º	3º
Tiempo original	41.455 seg	41.657 seg	41.676 seg
Tiempo	40.288 seg	40.604 seg	40.622 seg
Memoria driver	1500Mb	3GB	600Mb
Número de ejecutores	1	1	1
Memoria por ejecutor	1500Mb	600Mb	1500Mb
Núcleos	3	3	3
Factor de multiplicación	1	1	2
Nivel de paralelismo	3	3	6

Respecto a los peores tiempos, tenemos la misma situación que para el dataset anterior: más particiones implica más tiempo de ejecución.

3. **dic_100**

a) **word_count_level_2_jobs:** como en los casos anteriores, se obtiene mejora reduciendo las particiones del RDD que resulta de la suma de las apariciones de cada palabra:

	1º	2º	3º
Tiempo original	103.101 seg	105.313 seg	106.549 seg
Tiempo	100.226 seg	100.414 seg	100.882 seg
Memoria driver	600Mb	600Mb	1500Mb
Número de ejecutores	1	1	1
Memoria por ejecutor	1500Mb	1500Mb	600Mb
Núcleos	4	4	4
Factor de multiplicación	1	6	6
Nivel de paralelismo	4	24	24

b) **word_count_plus_jobs:** se obtienen mejoras de la misma forma que para dic_30 en esta aplicación:

	1º	2º	3º
Tiempo original	109.276 seg	111.194 seg	111.209 seg
Tiempo	101.964 seg	102.282 seg	102.588 seg
Memoria driver	1500Mb	3GB	1500Mb
Número de ejecutores	1	1	1
Memoria por ejecutor	600Mb	600Mb	1500Mb
Núcleos	4	4	4
Factor de multiplicación	2	2	1
Nivel de paralelismo	8	8	4

Volvemos a encontrarnos con la conclusión de los últimos resultados: más particiones implica más tiempo de ejecución.

4.9. Persistencia de datos

Debido a la naturaleza perezosa de la evaluación de los RDDs, si se quiere llamar más de una acción sobre el mismo RDD, Spark tendrá que volver a ejecutar el plan de transformación tantas veces como acciones se apliquen a los datos. Se puede evitar esta situación haciendo que Spark conserve el RDD. De esta forma, los nodos involucrados en su cálculo guardarán las particiones correspondientes. De hecho, para mayor tolerancia a fallos, se pueden replicar los datos en varios nodos para evitar que el fallo de un nodo implique la pérdida de sus datos persistidos y, en consecuencia, una nueva ejecución del plan lógico asociado.

Existen diferentes niveles de persistencia. La elección entre todos ellos dependerá de los recursos con los que se cuenta y los objetivos de rendimiento.

Nivel de almacenamiento	Espacio requerido	Eficiencia en CPU	Descripción
MEMORY_ONLY	Alto	Baja	Nivel por defecto. Se almacena el RDD en memoria. Si no hay espacio, se almacenará en caché lo posible y el resto de particiones se volverán a calcular cuando se requieran
MEMORY_AND_DISK	Alto	Media	Se almacena el RDD en memoria. Si no hay espacio, se almacenará en caché lo posible y el resto de particiones se almacenarán en disco para acceder a ellas cuando se requieran
DISK_ONLY	Bajo	Alta	Almacena los RDDs únicamente en disco
MEMORY_ONLY_N, MEMORY_AND_DISK_N, DISK_ONLY_N	-	-	Incluye en cada tipo de nivel el replicado de cada partición en en el número N de nodos del clúster (MEMORY_ONLY_2, MEMORY_AND_DISK_2, DISK_ANLY_2, MEMORY_ONLY_3, MEMORY_AND_DISK_3, DISK_ANLY_3, etc.)

Tabla 4.10: Niveles de persistencia

Para establecer el nivel de persistencia, se debe aplicar la función *.persist(Boolean, Boolean, Boolean, Boolean)* al RDD deseado. Los valores de tipo booleano que tiene esta función como argumento, determinarán el nivel de persistencia aplicado:

- rdd.persist(StorageLevel(False, True, False, False)) - MEMORY_ONLY
- rdd.persist(StorageLevel(True, False, False, False)) - DISK_ONLY
- rdd.persist(StorageLevel(True, True, False, False)) - MEMORY_AND_DISK
- rdd.persist(StorageLevel(False, True, False, False, N)) - MEMORY_ONLY_N
- rdd.persist(StorageLevel(True, False, False, False, N)) - DISK_ONLY_N
- rdd.persist(StorageLevel(True, True, False, False, N)) - MEMORY_AND_DISK_N

Hay que tener en cuenta que la simple llamada de persistencia sobre un RDD no fuerza la evaluación. Hay que realizar una acción para que la persistencia, al igual que las transformaciones, se lleven a cabo sobre los datos.

4.9.1. Resultados

Hasta ahora, se han ido observando puntos de mejora en los tiempos de ejecución importantes, como la estructura del dataset, el número de núcleos por ejecutor y, en algunos casos, la modificación de la cantidad de particiones de los RDDs generados. Teniendo en cuenta estos puntos de optimización, se han realizado ejecuciones con diversas estructuras de persistencia sobre la base de código que estamos utilizando. Al igual que para la sección 4.8.1, la idea de las ejecuciones ha sido ir persistiendo uno o más RDDs para observar posteriormente si afecta en el tiempo de ejecución.

Los resultados obtenidos en esta sección no evidencian una mejora de los tiempos de ejecución de las aplicaciones. Esto se debe a la apreciación anteriormente hecha, que la llamada a la persistencia no fuerza la evaluación, por lo que en la versión de las aplicaciones que solo tienen un *job* final no afectamos al plan lógico establecido. Por ello, se ha desarrollado otra versión utilizando como base de código el de las aplicaciones que incluyen acciones intermedias sobre los RDDs (ver sección 3.5). En este caso, se puede observar mejoras en el caso del dataset **dic_100** y, muy ligeras, en el caso de **dic_30**:

1. **dic_30**

   a) **word_count_level_2_jobs:** recordemos los mejores tiempos para esta aplicación y este dataset:

	1º	**2º**	**3º**
Tiempo	39.474 seg	39.679 seg	39.746 seg
Memoria driver	1500Mb	1500Mb	600Mb
Número de ejecutores	1	1	1
Memoria por ejecutor	3GB	600Mb	3GB
Núcleos	3	3	3

   Se añade a la aplicación original la persistencia del RDD *frequencies*,

```
data = sc.textFile(data_file)
words = data.flatMap(cleanData)

frequencies= words.filter(lambda x: x != '').map(lambda x: (x, 1)).
    reduceByKey(lambda a,b: a+b)
frequencies.persist(StorageLevel(Boolean, Boolean, Boolean, Boolean))
```

```
6       print('Example of words frequencies:', frequencies.collect()[1])
7
8       initialsFreqs = frequencies.map(lambda x: (x[0][0], (x[0], x[1])))
9       groupInitialFreqs = initialsFreqs.groupByKey()
10      print('Example of words frequencies group by initial:', groupInitialFreqs.
            collect()[1])
```

obteniendo:

	1º	2º	3º
Tiempo	39.228 seg	39.615 seg	39.667 seg
Memoria driver	1500Mb	600Mb	600Mb
Número de ejecutores	1	1	1
Memoria por ejecutor	1500Mb	3GB	1500Mb
Núcleos	3	3	3
Nivel persistencia	MEM_AND_DISK	MEM_ONLY	DISK_ONLY

b) **word_count_plus_jobs:** recordemos los mejores tiempos para esta aplicación y este dataset:

	1º	2º	3º
Tiempo	41.455 seg	41.657 seg	41.676 seg
Memoria driver	3GB	1500Mb	3GB
Número de ejecutores	1	1	1
Memoria por ejecutor	6GB	1500Mb	1500Mb
Núcleos	3	3	3

Se añade a la aplicación original la persistencia del RDD *reduceInitialFreqs*,

```
1       data = sc.textFile(data_file)
2
3       words = data.flatMap(cleanData)
4       frequencies= words.filter(lambda x: x != '').map(lambda x: (x, 1)).
            reduceByKey(lambda a,b: a+b)
5       print('Example of words frequencies:', frequencies.collect()[1])
6
7       initialsFreqs = frequencies.map(lambda x: (x[0][0], (x[0], x[1])))
8       groupInitialFreqs = initialsFreqs.groupByKey()
9       print('Example of words frequencies group by initial: ', groupInitialFreqs
            .collect()[1])
10
11      reduceInitialFreqs = initialsFreqs.map(lambda x: (x[0], x[1][1])).
            reduceByKey(lambda a,b: a + b)
12      reduceInitialFreqs.persist(StorageLevel(Boolean, Boolean, Boolean, Boolean
            ))
13
14      totalInfoFreqs = reduceInitialFreqs.join(initialsFreqs)
15      print('Example of result: ', totalInfoFreqs.collect()[1])
```

obteniendo:

	1º	2º	3º
Tiempo	39.63 seg	39.878 seg	40.127 seg
Memoria driver	600Mb	600Mb	1500Mb
Número de ejecutores	1	1	1
Memoria por ejecutor	600Mb	3GB	1500Mb
Núcleos	3	3	3
Nivel persistencia	MEM_AND_DISK	MEM_ONLY	MEM_AND_DISK

2. **dic_100**

a) **word_count_level_2:** recordemos los mejores tiempos para esta aplicación y este dataset:

	1º	2º	3º
Tiempo	103.101 seg	106.549 seg	107.511 seg
Memoria driver	600Mb	1500Mb	3GB
Número de ejecutores	1	1	1
Memoria por ejecutor	600Mb	6GB	6GB
Núcleos	4	4	4

Se añade a la aplicación original la persistencia del RDD *frequencies*,

```
    data = sc.textFile(data_file)

    words = data.flatMap(cleanData)
    frequencies= words.filter(lambda x: x != '').map(lambda x: (x, 1)).
        reduceByKey(lambda a,b: a+b)
    frequencies.persist(StorageLevel(Boolean, Boolean, Boolean, Boolean))
    print('Example of words frequencies:', frequencies.collect()[1])

    initialsFreqs = frequencies.map(lambda x: (x[0][0], (x[0], x[1])))
    groupInitialFreqs = initialsFreqs.groupByKey()
    print('Example of words frequencies group by initial: ', groupInitialFreqs
        .collect()[1])

    reduceInitialFreqs = initialsFreqs.map(lambda x: (x[0], x[1][1])).
        reduceByKey(lambda a,b: a + b)
    totalInfoFreqs = reduceInitialFreqs.join(initialsFreqs)
    print('Example of result: ', totalInfoFreqs.collect()[1])
```

obteniendo:

	1º	2º	3º
Tiempo	99.969 seg	101.962 seg	103.632 seg
Memoria driver	1500Mb	1500Mb	3GB
Número de ejecutores	1	1	1
Memoria por ejecutor	600Mb	1500Mb	3GB
Núcleos	4	4	4
Nivel persistencia	MEM_ONLY	MEM_AND_DISK	DISK_ONLY

b) **word_count_plus:** recordemos los mejores tiempos para esta aplicación y este dataset:

	1º	2º	3º
Tiempo	109.276 seg	111.209 seg	111.49 seg
Memoria driver	600Mb	3GB	1500Mb
Número de ejecutores	1	1	1
Memoria por ejecutor	3GB	600Mb	600Mb
Núcleos	4	4	4

Se muestra el caso de añadir la persistencia al RDD *initialsFreqs*, al que corresponde el mejor tiempo. El segundo y tercer puesto lo obtiene la aplicación que persiste el RDD *reduceInitialFreqs*, que será análogo al código que se va a mostrar pero cambiando de posición la función *persist()*:

```
    data = sc.textFile(data_file)

    words = data.flatMap(cleanData)
    frequencies= words.filter(lambda x: x != '').map(lambda x: (x, 1)).
        reduceByKey(lambda a,b: a+b)

    initialsFreqs = frequencies.map(lambda x: (x[0][0], (x[0], x[1])))
    initialsFreqs.persist(StorageLevel(Boolean, Boolean, Boolean, Boolean))
    groupInitialFreqs = initialsFreqs.groupByKey()
    print('Example of words frequencies group by initial: ', groupInitialFreqs
        .collect()[1])

    reduceInitialFreqs = initialsFreqs.map(lambda x: (x[0], x[1][1])).
        reduceByKey(lambda a,b: a + b)
```

```
12      totalInfoFreqs = reduceInitialFreqs.join(initialsFreqs)
13      print('Example of result: ', totalInfoFreqs.collect()[1])
```

obteniendo:

	1º	2º	3º
Tiempo	104.160 seg	105.2435 seg	105.919 seg
Memoria driver	1500Mb	1500Mb	1500Mb
Número de ejecutores	1	1	1
Memoria por ejecutor	600Mb	600Mb	1500Mb
Núcleos	4	4	4
Nivel persistencia	DISK_ONLY	MEM_AND_DISK	MEM_AND_DISK

4.10. Control del *shuffle*

Trabajando con RDDs de pares, a menudo, necesitamos ubicar las apariciones de claves concretas para realizar ciertas operaciones. Esto implica, en el peor de los casos, buscar en cada partición. En los puntos anteriores ya se observa alguna manera de evitar esta costosa operación o, al menos, reducir su peso:

- Persistiendo RDDs intermedios que serán recalculados en futuras acciones.
- Establecer un mismo particionador en los RDDs implicados en un *join*().
- Controlar el sesgo de datos para reducir la cantidad de datos que se transfieren a través del clúster.

Otra forma de disminuir la cantidad de datos involucrados en el *shuffle* es prevenir el uso de la operación *groupByKey()*. Esta transformación *wide* provoca el mezclado de todos los datos del RDD en cuestión. En su lugar, se usará la operación *reduceByKey()*, del mismo tipo que la anterior, la cual sólo realizará la transmisión de los datos resultantes de la agrupación por claves en cada partición.

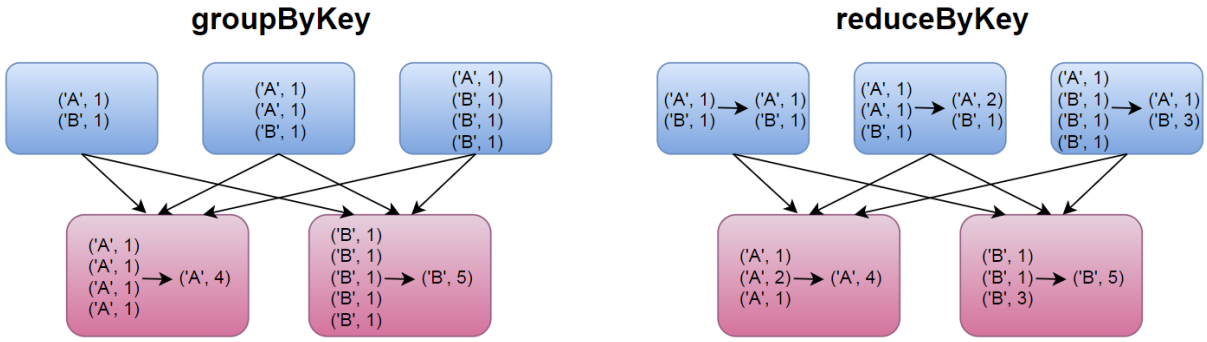


Figura 4.20: Diferencia en la cantidad de datos transferidos entre las operaciones GroupByKey y ReduceByKey

En general, evitar el *shuffle* hace nuestras aplicaciones más rápidas. Sin embargo, hay situaciones excepcionales en las que conviene que se lleve a cabo el *shuffle*. Una de ellas es que las particiones que se establezcan en la lectura de los datos tengan demasiados registros. En esta situación, forzar el *shuffle* nos va a permitir reparticionar los datos, por ejemplo, invocando *repartition* para tener una mayor cantidad de particiones con menos registros. La otra situación en la que generar *shuffle* puede evitarnos problemas es al utilizar las funciones de reducción o agregación regulares de Spark, donde en las que cada

tarea implicada envían los valores resultantes al *driver*. Así, si se tienen muchos registros distribuidos uniformemente en una gran cantidad de claves se podría producir un cuello de botella al enviar los datos. Para reducir la cantidad de particiones que envían datos al controlador, se puede optar por realizar la transformación *reduceByKey* o *aggregateByKey* previamente. Se trata de una situación similar a la vista en la comparación de las funciones *groupByKey* y *reduceByKey* pero aquí estamos planteando el forzar el uso de la operación en vez de ser una alternativa más eficiente. Estas funciones tienen la propiedad de que las particiones encargadas de realizar las operaciones se comunican entre si un número logarítmico de veces. Los datos se mezclan en un un pequeño número de ejecutores antes de ser enviados al *driver*, reduciendo el trabajo de recopilación para el controlador.

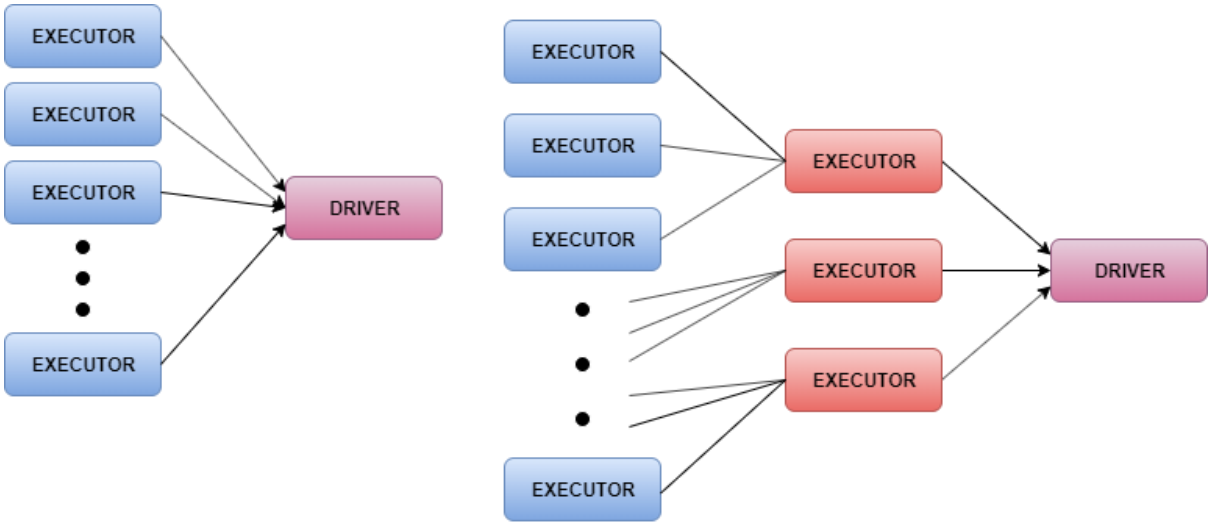


Figura 4.21: Uso de *reduceByKey* o *aggregateByKey* para dividir el conjunto de datos en un número menor de particiones

4.10.1. Resultados

Las aplicaciones de este apartado se han generado reemplazando la operación *groupByKey()* por *reduceByKey()*. A continuación, se muestra el ejemplo de la aplicación **word_count_level_2_reduce**, en la que se aplica este cambio:

```
1       data = sc.textFile(data_file)
2       words = data.flatMap(cleanData)
3       frequencies= words.filter(lambda x: x != '').map(lambda x: (x, 1)).\textsl{
            reduceByKey}(lambda a,b: a+b)
4
5       initialsFreqs = frequencies.map(lambda x: (x[0][0], (x[0], x[1])))
6       groupInitialFreqs = initialsFreqs.\textsl{reduceByKey}(lambda a,b: a+b)
7       print('Example of words frequencies group by initial:', groupInitialFreqs.collect()
            [1])
```

Como en la sección anterior, las ejecuciones sobre las aplicaciones con un solo *job* final no aportan optimización sobre los tiempos de ejecución.

Para las versiones con más de una acción, únicamente se obtienen mejores resultados para el dataset más grande. Para los otros dos casos, se obtienen tiempos muy similares a los originales.

dic_100

1. **word_count_level_2:**

	1º	2º	3º
Tiempo	99.378 seg	99.57 seg	100.516 seg
Memoria driver	1500Mb	3GB	1500Mb
Número de ejecutores	1	1	1
Memoria por ejecutor	1500Mb	3GB	600Mb
Núcleos	4	4	4

2. **word_count_plus:**

	1º	2º	3º
Tiempo	103.431 seg	104.607 seg	106.471 seg
Memoria driver	6GB	4g	3GB
Número de ejecutores	1	1	1
Memoria por ejecutor	1500Mb	1500Mb	1500Mb
Núcleos	4	4	4

4.11. Resultados conjuntos

Para las aplicaciones con una sola acción final, se ha logrado reducir el tiempo de ejecución únicamente con los ajustes de particiones. Sin embargo, para las aplicaciones con más de un *job*, las mejoras se han alcanzado mediante la modificación del número de particiones, la persistencia de los RDDs, el uso de la operación *reduceByKey* en lugar de *groupByKey* (correspondiente al *shuffle*). Por ello, resulta interesante agrupar estas mejoras y comprobar si todas juntas afectarían en mayor medida al tiempo de ejecución de las aplicaciones. Se han seleccionado las aplicaciones que han obtenido mejores tiempos y se han desarrollado diferentes versiones combinando los puntos de mejora. De esta forma, se tienen cuatro posibilidades de incorporación en las aplicaciones: las tres mejoras, ajuste de particiones con persistencia, *shuffle* con persistencia y *shuffle* con ajuste de particiones.

Las aplicaciones que se van a integrar en versiones conjuntas son:

- Word Count Level 2:
  - Reparticionando el RDD *frequencies*.
  - Persistiendo en todos los niveles el RDD *frequencies*.
  - Reemplazo de la operación *groupByKey()* por *reduceByKey()*
- Word Count Plus:
  - Reparticionando el RDD *frequencies* y los RDDs involucrados en el *join* final.
  - Persistiendo en disco el RDD *initialsFreqs*.
  - Persistiendo en memoria y disco el RDD *reduceInitialFreqs*.
  - Reemplazo de la operación *groupByKey()* por *reduceByKey()*

Veamos los resultados obtenidos:

1. **dic_15**

   a) **word_count_level_2_jobs:** únicamente se recoge un resultado que disminuye el mejor tiempo. Este resultado corresponde a la aplicación que combina la persistencia en disco del RDD *frequencies* y la repartición de este mismo RDD en dos particiones (que es el caso para el

que se había registrado el mejor tiempo hasta ahora: 23.513 segundos). En este caso, vemos que añadiendo la persistencia al particionado se logra mejorar ligeramente el tiempo de ejecución:

	1º
Tiempo	23.263 seg
Memoria driver	1500Mb
Número de ejecutores	1
Memoria por ejecutor	1500Mb
Núcleos	4
Factor de multiplicación	0.5
Nivel de paralelismo	2
Nivel persistencia	DISK_ONLY

b) **word_count_plus_jobs:** se registra disminución de tiempo para la aplicación que persiste en memoria y disco el RDD *reduceInitialFreqs* y reparticiona el RDD *frequencies* y los RDDs involucrados en el *join* final. Hasta ahora, se había conseguido reducir el tiempo a 24.797 segundos, con el ajuste de particiones. El tercer mejor tiempo también recoge mejora relativa al *shuffle*, por lo que obtenemos uno de los mejores tiempos con la aplicación de todas las mejoras vistas:

	1º	2º	3º
Tiempo	24.23 seg	24.678 seg	24.688 seg
Memoria driver	600Mb	1500Mb	3GB
Número de ejecutores	1	1	1
Memoria por ejecutor	3GB	3GB	1500Mb
Núcleos	4	4	4
Factor de multiplicación	0.5	0.5	0.5
Nivel de paralelismo	2	2	2
Nivel persistencia	MEM_AND_DISK	MEM_AND_DISK	MEM_AND_DISK

2. **dic_30**

a) **word_count_level_2_jobs:** no se registran resultados menores que los 38.803 segundos obtenidos con el ajuste de particiones.

b) **word_count_plus_jobs:** tampoco conseguimos mejoría respecto a los 39.63 segundos alcanzados mediante la persistencia

3. **dic_100**

a) **word_count_level_2_jobs:** el mejor tiempo hasta ahora para este caso se obtenía con el ajuste de las particiones del RDD *frequencies*, con 100.226 segundos. Mediante la repartición del RDD *frequencies*, la persistencia en memoria y disco de este RDD y el uso de *reduceByKey()* en lugar de *groupByKey*, se registra un tiempos menor:

	1º
Tiempo	99.387 seg
Memoria driver	3GB
Número de ejecutores	1
Memoria por ejecutor	600Mb
Núcleos	4
Factor de multiplicación	5
Nivel de paralelismo	20
Nivel persistencia	MEM_AND_DISK

b) **word_count_plus_jobs:** igual que para **word_count_level_2_jobs:** , con el reparticionado del RDD *frequencies*, se obtenían 101.964 segundos como mejore resultado. Conseguí-

mos una ligera reducción de este tiempo mediante la persistencia en memoria y disco del RDD *reduceInitialFreqs* y la repartición de los RDDs: frequencies, initialsFreqs y reduceInitialFreqs.

	1º
Tiempo	101.65 seg
Memoria driver	3GB
Número de ejecutores	1
Memoria por ejecutor	600Mb
Núcleos	4
Factor de multiplicación	2.5
Nivel de paralelismo	10
Nivel persistencia	MEM_AND_DISK

Capítulo 5

Conclusiones

Recordemos que el presente trabajo tiene como propósito general el análisis de la capacidad de optimización del rendimiento y de los recursos en las aplicaciones Spark. Como objetivos específicos se incluían el estudio detallado de la herramienta y el enfoque hacia las buenas prácticas para crear nuestras aplicaciones.

Apache Spark es una herramienta con una gran comunidad de colaboración y un gran uso por parte de empresas que requieren del procesamiento de grandes conjuntos de datos. Por ello, es importante entender la herramienta y las posibilidades que brinda a la hora de ajustar los procesos a las necesidades específicas. Este conocimiento va a permitir orientar el desarrollo de las aplicaciones a los recursos disponibles y casos de uso concretos. Aunque nuestro marco de desarrollo no sea equivalente al de una gran empresa tecnológica, se ha pretendido realizar un análisis escalable a una estructura de mayor alcance y capacidad.

Nuestro primer punto de estudio, la configuración de los recursos, ha aportado varios aspectos importantes a tener en cuenta. Hemos podido comprobar como la elección de los núcleos determina en gran medida el tiempo de procesado de los datos, ya que marcan el nivel principal de paralelismo que vamos a tener en nuestro procesado. Hay que tener en cuenta la limitación que se ha tenido respecto a los recursos del clúster utilizado durante el desarrollo. Ya que solo podíamos disponer de uno o dos ejecutores según el número de núcleos, la elección de estas dos variables se ha reducido a un ejecutor y tres o cuatro núcleos, puesto que la combinación de dos ejecutores con dos núcleos cada uno no ha resultado eficiente respecto al rendimiento. Como era de esperar, también se descartan como ejecuciones de interés las establecidas con un solo núcleo por ejecutor, al no aprovechar el paralelismo disponible. Se ha podido comprobar que para los datasets de menor y mayor tamaño (1.5Gb y 10Gb aproximadamente) la mejor elección son cuatro núcleos por ejecutor. Sin embargo, para el dataset de alrededor de 3Gb la elección óptima son tres núcleos. Esto evidencia la importancia del tamaño de nuestros datos a la hora de ajustar los recursos y dividir el trabajo.

Respecto a la memoria asignada tanto al driver como a los ejecutores, no se han generado resultados que evidencien una combinación preferible. En las figuras 4.20 a 4.28 se puede ver la disparidad de asignación de memoria para los mejores resultados por dataset y aplicación. El único caso en el que se puede ver una cierta relación entre memoria asignada y optimización es para el dataset dic_100 y la aplicación Word Count Plus. Sin embargo, en los siguientes tiempos también se observa la presencia de 3Gb para la memoria del driver y 3Gb o 6Gb para la memoria del ejecutor. Se puede relacionar esta falta de conclusiones respecto a la asignación de memoria con los recursos de los que se dispone en nuestro clúster. Habría que analizar la misma casuística en un clúster con una cantidad de recursos mayor.

El análisis de la cantidad de particiones establecidas para los RDDs han evidenciado que es un punto relevante a la hora de optimizar nuestros procesos. Para todos los datasets y la mayoría de las aplica-

ciones, se han conseguido mejores resultados modificando el particionado establecido por la lógica de Spark. En el caso de la aplicación Word Count, las mejoras obtenida para dic_15 y dic_30 se corresponden con el aumento del número de particiones que se establecen en la creación del primer RDD (en la lectura de los datos). En cambio, para las aplicaciones Word Count Level 2 y Word Count Plus se logran mejores tiempos reduciendo significativamente esta cantidad y realizando el reparticionado a RDDs intermedios en la aplicación. De hecho, al aumentar el número de particiones se obtienen tiempos notablemente mayores. De estos resultados, se puede deducir que el proceso de lectura de los datos es una operación con menor coste computacional que el resto de operaciones, ya que requiere dividir los datos en un mayor grado de paralelismo para tardar menos en procesarlos. Sin embargo, para operaciones intermedias conviene que el grado de paralelismo sea mucho menor: menos particiones y más grandes.

En cuanto al papel de la persistencia de los RDDs, como se menciona en la sección 4.9.1, la persistencia sin una acción posterior no va a afectar al plan lógico establecido, por lo que no afectará en los resultados. Por ello, solo encontramos resultados favorables en los casos de las aplicaciones con más de un *job*, aunque no en el caso del dataset dic_15. Obteniendo estas mejoras, podemos asegurar que el uso de la persistencia afecta al rendimiento de nuestras aplicaciones, aunque no podemos sacar conclusiones respecto al nivel de paralelismo. De la misma forma que con la asignación de memoria, encontramos entre los mejores resultados la mayoría de posibilidades. Por ello, apuntamos a los recursos existentes en nuestro clúster para la ausencia de capacidad de elección frente al nivel de persistencia preferible.

Los resultados referentes al control del *shuflle* son claros: obtenemos un mejor rendimiento utilizando la operación *reduceByKey()* en lugar de la operación *groupByKey()*. Además, se han mencionado otras situaciones favorables que se relacionan directamente con el *shuffle*. La persistencia es una de ellas, puesto que persistiendo un RDD que conlleva mezclado de datos en su cálculo, evitamos repetir dicho proceso de *shuffle*. También lo podemos observar con algunos de los resultados obtenidos en la sección 4.8, para la aplicación Word Count Plus. En esta, se recogen mejores resultados cuando se establece la misma cantidad de particiones (y mismo particionador) a los RDDs que posteriormente se juntan mediante la función *join*.

Como último punto de resultados, se recogía una pregunta muy propicia: si incorporamos todas las situaciones de mejora observadas en una misma aplicación, ¿lograremos mejores resultados?. Aparentemente no. Aunque se han recogido algún tiempo menor (en el caso de dic_15 y dic_100) no parece ser algo determinante ya que son tiempos muy próximos a los ya obtenidos. De nuevo, sería conveniente realizar estas pruebas de mejoras aisladas y conjuntas en un clúster con mayores recursos y, posiblemente, con conjuntos de datos más grandes.

Se han dado puntos en los que no podemos concluir los escenarios favorables. Si nos centramos en los resultados destacables para nuestro marco de trabajo, podemos verificar que los objetivos inicialmente establecidos han sido viables y convenientes. Se han determinado puntos de mejora y aspectos de relevancia, como la elección de la estructura de los datos, para la optimización de las aplicaciones Spark.

run_job.py

```
import sys
import os
from datetime import datetime
import shutil

def main(mode, log_dir, rep, internal_param, data_f, script):
    dateTime = datetime.now()

    if not os.path.exists("/home/janira/"+str(script)+"Results"+str(data_f)):
      os.makedirs("/home/janira/"+str(script)+"Results"+str(data_f))

    if mode == 'test':
        print('test',internal_param,'@', data_f, '+', rep, log_dir)

    elif mode == 'run':
      resultsFile = open("/home/janira/"+str(script)+"Results"+str(data_f)+"/result-"+
          str(internal_param[1:])+".txt","a")
      print('exec',internal_param,'@', data_f, '+', rep, log_dir)
      resultsFile.write("-----------------------------------\n")
      resultsFile.write("-----------------------------------\n")
      resultsFile.write("------------DATASET "+ str(data_f) + str(dateTime)+"-- REP
          NUMBER " + str(rep) +"\n")
      resultsFile.write("-----------------------------------\n")
      resultsFile.write("-----------------------------------\n\n")

      try:
        app = str(internal_param[1])
        app_id = eval(app)(internal_param, data_f)
        print('mv', app_id,'to', log_dir)
        shutil.move('/opt/spark/current/logs/'+app_id, log_dir+"/"+str(app)+"_"+app_id
            )
        os.system('python3 /home/janira/logscript.py '+str(app_id)+' '+str(log_dir)+'
            '+str(data_f)+' '+str(internal_param)+' '+str(script))
      except:
        print("Configuration error: "+str(internal_param))
    else:
        print('mode error, select (test|run)')

if __name__ == '__main__':
    sep = sys.argv.index('+')
    internal_param = sys.argv[:sep]
    mode, log_dir, rep, data_f, script = sys.argv[sep+1:]
    print ('internal parameters', internal_param)
    print ('general parameters', mode, log_dir, rep)
    print ('data file', data_f)

    if "words_to_dic" in data_f:
      from config_words_to_dic import *
    elif "dic_to_words" in data_f:
      from config_dic_to_words import *
    elif "dic" in data_f:
      from config_dic import *
      from repartition_dic import *
      from repartition_jobs_dic import *
      from persist_dic import *
      from skew_dic import *
      from shuffle_dic import *
    else:
      from config_words import *
      from repartition_words import *
      from skew_words import *

    if not os.path.exists(log_dir):
            os.makedirs(log_dir)

    main(mode, log_dir, rep, internal_param, data_f, script)
```

config_scripts.py

```
from pyspark import SparkContext, SparkConf
  import sys
  import string
  import re

  def cleanData(x):
      x = re.sub('[^a-zA-Z0-9 ]', "", x.lower())
      line = x.split(" ")
      return line

  def word_count(internal_param, data_file):
      try:
        conf = SparkConf().setMaster("spark://dana:7077").setAppName(internal_param[1]).
            setAll([('spark.driver.cores', internal_param[2]), ('spark.driver.memory',
            internal_param[3]), ('spark.executor.instances', internal_param[4]), ('spark
            .executor.memory', internal_param[5]), ('spark.executor.cores',
            internal_param[6])])
        sc = SparkContext(conf=conf, pyFiles=['run_app.py', 'config_dic.py', 'run_job.py
            '])

        data = sc.textFile(data_file)
        words = data.flatMap(cleanData)
        frequencies= words.filter(lambda x: x != '').map(lambda x: (x, 1)).reduceByKey(
            lambda a,b: a+b)
        print('Example of words frequencies:', frequencies.collect()[1])

        app_id = sc.applicationId
        print('app end: ', app_id)

        sc.stop()
        return app_id
      except:
        print("Configuration error: "+str(internal_param))
        sc.stop()

  def word_count_level_2_stages(internal_param, data_file):
      try:
        conf = SparkConf().setMaster("spark://dana:7077").setAppName(internal_param[1]).
            setAll([('spark.driver.cores', internal_param[2]), ('spark.driver.memory',
            internal_param[3]), ('spark.executor.instances', internal_param[4]), ('spark
            .executor.memory', internal_param[5]), ('spark.executor.cores',
            internal_param[6])])
        sc = SparkContext(conf=conf, pyFiles=['run_app.py', 'config_dic.py', 'run_job.py
            '])

        data = sc.textFile(data_file)
        words = data.flatMap(cleanData)
        frequencies= words.filter(lambda x: x != '').map(lambda x: (x, 1)).reduceByKey(
            lambda a,b: a+b)

        initialsFreqs = frequencies.map(lambda x: (x[0][0], (x[0], x[1])))
        groupInitialFreqs = initialsFreqs.groupByKey()
        print('Example of words frequencies group by initial:', groupInitialFreqs.
            collect()[1])

        app_id = sc.applicationId

        print('app end: ', app_id)

        sc.stop()
        return app_id
      except:
        print("Configuration error: "+str(internal_param))
        sc.stop()

  def word_count_plus_stages(internal_param, data_file):
      try:
```

```
55          conf = SparkConf().setMaster("spark://dana:7077").setAppName(internal_param[1]).
                setAll([('spark.driver.cores', internal_param[2]), ('spark.driver.memory',
                internal_param[3]), ('spark.executor.instances', internal_param[4]), ('spark
                .executor.memory', internal_param[5]), ('spark.executor.cores',
                internal_param[6])])
56          sc = SparkContext(conf=conf, pyFiles=['run_app.py', 'config_dic.py', 'run_job.py
                '])
57
58          data = sc.textFile(data_file)
59          words = data.flatMap(cleanData)
60          frequencies= words.filter(lambda x: x != '').map(lambda x: (x, 1)).reduceByKey(
                lambda a,b: a+b)
61
62          initialsFreqs = frequencies.map(lambda x: (x[0][0], (x[0], x[1])))
63          groupInitialFreqs = initialsFreqs.groupByKey()
64
65          reduceInitialFreqs = initialsFreqs.map(lambda x: (x[0], x[1][1])).reduceByKey(
                lambda a,b: a + b)
66          totalInfoFreqs = reduceInitialFreqs.join(initialsFreqs)
67          print('Example of result: ', totalInfoFreqs.collect()[1])
68
69          app_id = sc.applicationId
70          print('app end: ', app_id)
71
72          sc.stop()
73          return app_id
74        except:
75          print("Configuration error: "+str(internal_param))
76          sc.stop()
77
78
79    def word_count_level_2_jobs(internal_param, data_file):
80        try:
81          conf = SparkConf().setMaster("spark://dana:7077").setAppName(internal_param[1]).
                setAll([('spark.driver.cores', internal_param[2]), ('spark.driver.memory',
                internal_param[3]), ('spark.executor.instances', internal_param[4]), ('spark
                .executor.memory', internal_param[5]), ('spark.executor.cores',
                internal_param[6])])
82          sc = SparkContext(conf=conf, pyFiles=['run_app.py', 'config_dic.py', 'run_job.py
                '])
83
84          data = sc.textFile(data_file)
85          words = data.flatMap(cleanData)
86          frequencies= words.filter(lambda x: x != '').map(lambda x: (x, 1)).reduceByKey(
                lambda a,b: a+b)
87          print('Example of words frequencies:', frequencies.collect()[1])
88
89          initialsFreqs = frequencies.map(lambda x: (x[0][0], (x[0], x[1])))
90          groupInitialFreqs = initialsFreqs.groupByKey()
91          print('Example of words frequencies group by initial: ', groupInitialFreqs.
                collect()[1])
92
93          app_id = sc.applicationId
94
95          print('app end: ', app_id)
96
97          sc.stop()
98          return app_id
99        except:
100         print("Configuration error: "+str(internal_param))
101         sc.stop()
102
103   def word_count_plus_jobs(internal_param, data_file):
104       try:
105         conf = SparkConf().setMaster("spark://dana:7077").setAppName(internal_param[1]).
                setAll([('spark.driver.cores', internal_param[2]), ('spark.driver.memory',
                internal_param[3]), ('spark.executor.instances', internal_param[4]), ('spark
                .executor.memory', internal_param[5]), ('spark.executor.cores',
                internal_param[6])])
106         sc = SparkContext(conf=conf, pyFiles=['run_app.py', 'config_dic.py', 'run_job.py
                '])
107
108         data = sc.textFile(data_file)
109         words = data.flatMap(cleanData)
110         frequencies= words.filter(lambda x: x != '').map(lambda x: (x, 1)).reduceByKey(
                lambda a,b: a+b)
111         print('Example of words frequencies:', frequencies.collect()[1])
112
113         initialsFreqs = frequencies.map(lambda x: (x[0][0], (x[0], x[1])))
114         groupInitialFreqs = initialsFreqs.groupByKey()
115         print('Example of words frequencies group by initial: ', groupInitialFreqs.
                collect()[1])
116
117         reduceInitialFreqs = initialsFreqs.map(lambda x: (x[0], x[1][1])).reduceByKey(
                lambda a,b: a + b)
118         totalInfoFreqs = reduceInitialFreqs.join(initialsFreqs)
119         print('Example of result: ', totalInfoFreqs.collect()[1])
```

```
120
121           app_id = sc.applicationId
122           print('app end: ', app_id)
123
124           sc.stop()
125           return app_id
126         except:
127           print("Configuration error: "+str(internal_param))
128           sc.stop()
```

repartition_scripts.py

```
1  from pyspark import SparkContext, SparkConf
2  import sys
3  import string
4  import re
5
6  def cleanData(x):
7    x = re.sub('[^a-zA-Z0-9 ]', "", x.lower())
8    line = x.split(" ")
9    return line
10
11 def word_count_repartition_n(internal_param, data_file):
12   try:
13     conf = SparkConf().setMaster("spark://dana:7077").setAppName(internal_param[1]).
         setAll([('spark.driver.cores', internal_param[2]), ('spark.driver.memory',
         internal_param[3]), ('spark.executor.instances', internal_param[4]), ('spark.
         executor.memory', internal_param[5]), ('spark.executor.cores', internal_param
         [6])])
14     sc = SparkContext(conf=conf, pyFiles=['run_app.py', 'repartition_dic.py', 'run_job
         .py'])
15
16     data = sc.textFile(data_file)
17
18     words = data.flatMap(cleanData).repartition(int(int(sc.getConf().get("spark.
         executor.cores")) * int(sc.getConf().get("spark.executor.instances")) * float(
         internal_param[7])))
19     frequencies= words.filter(lambda x: x != '').map(lambda x: (x, 1)).reduceByKey(
         lambda a,b: a+b)
20     print('Example of words frequencies:', frequencies.collect()[1])
21
22     app_id = sc.applicationId
23     print('app end: ', app_id)
24
25     sc.stop()
26     return app_id
27   except:
28     print("Configuration error: "+str(internal_param))
29     sc.stop()
30
31 def word_count_level_2_repartition_n(internal_param, data_file):
32   try:
33     conf = SparkConf().setMaster("spark://dana:7077").setAppName(internal_param[1]).
         setAll([('spark.driver.cores', internal_param[2]), ('spark.driver.memory',
         internal_param[3]), ('spark.executor.instances', internal_param[4]), ('spark.
         executor.memory', internal_param[5]), ('spark.executor.cores', internal_param
         [6])])
34     sc = SparkContext(conf=conf, pyFiles=['run_app.py', 'repartition_dic.py', 'run_job
         .py'])
35
36     data = sc.textFile(data_file).repartition(int(int(sc.getConf().get("spark.executor
         .cores")) * int(sc.getConf().get("spark.executor.instances")) * float(
         internal_param[7])))
37
38     words = data.flatMap(cleanData)
39     frequencies= words.filter(lambda x: x != '').map(lambda x: (x, 1)).reduceByKey(
         lambda a,b: a+b)
40     repartfrequencies = frequencies.repartition(int(int(sc.getConf().get("spark.
         executor.cores")) * int(sc.getConf().get("spark.executor.instances")) * float(
         internal_param[7])))
41
42     initialsFreqs = repartfrequencies.map(lambda x: (x[0][0], (x[0], x[1])))
43     groupInitialFreqs = initialsFreqs.groupByKey()
44     print('Example of words frequencies group by initial:', groupInitialFreqs.collect
         ()[1])
45
46     app_id = sc.applicationId
47
48     print('app end: ', app_id)
```

```
49
50        sc.stop()
51        return app_id
52      except:
53        print("Configuration error: "+str(internal_param))
54        sc.stop()
55
56    def word_count_plus_repartition_n(internal_param, data_file):
57      try:
58        conf = SparkConf().setMaster("spark://dana:7077").setAppName(internal_param[1]).
              setAll([('spark.driver.cores', internal_param[2]), ('spark.driver.memory',
              internal_param[3]), ('spark.executor.instances', internal_param[4]), ('spark.
              executor.memory', internal_param[5]), ('spark.executor.cores', internal_param
              [6])])
59        sc = SparkContext(conf=conf, pyFiles=['run_app.py', 'repartition_dic.py', 'run_job
              .py'])
60
61        data = sc.textFile(data_file)
62        repartitionData = data.repartition(int(int(sc.getConf().get("spark.executor.cores"
              )) * int(sc.getConf().get("spark.executor.instances")) * float(internal_param
              [7])))
63
64        words = repartitionData.flatMap(cleanData)
65        frequencies= words.filter(lambda x: x != '').map(lambda x: (x, 1)).reduceByKey(
              lambda a,b: a+b)
66        repartFrequencies = frequencies.repartition(int(int(sc.getConf().get("spark.
              executor.cores")) * int(sc.getConf().get("spark.executor.instances")) * float(
              internal_param[7])))
67
68        initialsFreqs = repartFrequencies.map(lambda x: (x[0][0], (x[0], x[1])))
69        groupInitialFreqs = initialsFreqs.groupByKey()
70
71        reduceInitialFreqs = initialsFreqs.map(lambda x: (x[0], x[1][1])).reduceByKey(
              lambda a,b: a + b).repartition(int(int(sc.getConf().get("spark.executor.cores"
              )) * int(sc.getConf().get("spark.executor.instances")) * float(internal_param
              [7])))
72        totalInfoFreqs = reduceInitialFreqs.join(initialsFreqs)
73        print('Example of result: ', totalInfoFreqs.collect()[1])
74
75        app_id = sc.applicationId
76        print('app end: ', app_id)
77
78        sc.stop()
79        return app_id
80      except:
81        print("Configuration error: "+str(internal_param))
82        sc.stop()
```

persist_scripts.py

```
1   from pyspark import SparkContext, SparkConf, StorageLevel
2   import sys
3   import string
4   import time
5   import re
6
7   def cleanData(x):
8     x = re.sub('[^a-zA-Z0-9 ]', "", x.lower())
9     line = x.split(" ")
10    return line
11
12  def word_count_persist_mem_only(internal_param, data_file):
13    try:
14      conf = SparkConf().setMaster("spark://dana:7077").setAppName(internal_param[1]).
          setAll([('spark.driver.cores', internal_param[2]), ('spark.driver.memory',
          internal_param[3]), ('spark.executor.instances', internal_param[4]), ('spark.
          executor.memory', internal_param[5]), ('spark.executor.cores', internal_param
          [6])])
15      sc = SparkContext(conf=conf, pyFiles=['run_app.py', 'persist_dic.py', 'run_job.py'
          ])
16
17      data = sc.textFile(data_file)
18      data.persist(StorageLevel(False, True, False, False))
19
20      words = data.flatMap(cleanData)
21      frequencies= words.filter(lambda x: x != '').map(lambda x: (x, 1)).reduceByKey(
          lambda a,b: a+b)
22      print('Example of words frequencies:', frequencies.collect()[1])
23
24      app_id = sc.applicationId
25      print('app end: ', app_id)
26
27      sc.stop()
28      return app_id
29    except:
30      print("Configuration error: "+str(internal_param))
31      sc.stop()
32
33  def word_count_persist_mem_and_disk(internal_param, data_file):
34    try:
35      conf = SparkConf().setMaster("spark://dana:7077").setAppName(internal_param[1]).
          setAll([('spark.driver.cores', internal_param[2]), ('spark.driver.memory',
          internal_param[3]), ('spark.executor.instances', internal_param[4]), ('spark.
          executor.memory', internal_param[5]), ('spark.executor.cores', internal_param
          [6])])
36      sc = SparkContext(conf=conf, pyFiles=['run_app.py', 'persist_dic.py', 'run_job.py'
          ])
37
38      data = sc.textFile(data_file)
39      data.persist(StorageLevel(True, True, False, False))
40
41      words = data.flatMap(cleanData)
42      frequencies= words.filter(lambda x: x != '').map(lambda x: (x, 1)).reduceByKey(
          lambda a,b: a+b)
43      print('Example of words frequencies:', frequencies.collect()[1])
44
45      app_id = sc.applicationId
46      print('app end: ', app_id)
47
48      sc.stop()
49      return app_id
50    except:
51      print("Configuration error: "+str(internal_param))
52      sc.stop()
53
54  def word_count_persist_disk_only(internal_param, data_file):
55    try:
```

```
56        conf = SparkConf().setMaster("spark://dana:7077").setAppName(internal_param[1]).
              setAll([('spark.driver.cores', internal_param[2]), ('spark.driver.memory',
              internal_param[3]), ('spark.executor.instances', internal_param[4]), ('spark.
              executor.memory', internal_param[5]), ('spark.executor.cores', internal_param
              [6])])
57        sc = SparkContext(conf=conf, pyFiles=['run_app.py', 'persist_dic.py', 'run_job.py'
              ])
58
59        data = sc.textFile(data_file)
60        data.persist(StorageLevel(True, False, False, False))
61
62        words = data.flatMap(cleanData)
63        frequencies= words.filter(lambda x: x != '').map(lambda x: (x, 1)).reduceByKey(
              lambda a,b: a+b)
64        frequencies
65        print('Example of words frequencies:', frequencies.collect()[1])
66
67        app_id = sc.applicationId
68        print('app end: ', app_id)
69
70        sc.stop()
71        return app_id
72      except:
73        print("Configuration error: "+str(internal_param))
74        sc.stop()
75
76    def word_count_level_2_mem_only_frequencies(internal_param, data_file):
77      try:
78        conf = SparkConf().setMaster("spark://dana:7077").setAppName(internal_param[1]).
              setAll([('spark.driver.cores', internal_param[2]), ('spark.driver.memory',
              internal_param[3]), ('spark.executor.instances', internal_param[4]), ('spark.
              executor.memory', internal_param[5]), ('spark.executor.cores', internal_param
              [6])])
79        sc = SparkContext(conf=conf, pyFiles=['run_app.py', 'persist_dic.py', 'run_job.py'
              ])
80
81        data = sc.textFile(data_file)
82        words = data.flatMap(cleanData)
83        frequencies= words.filter(lambda x: x != '').map(lambda x: (x, 1)).reduceByKey(
              lambda a,b: a+b)
84        frequencies.persist(StorageLevel(False, True, False, False))
85
86        initialsFreqs = frequencies.map(lambda x: (x[0][0], (x[0], x[1])))
87        groupInitialFreqs = initialsFreqs.groupByKey()
88        print('Example of words frequencies group by initial:', groupInitialFreqs.collect
              ()[1])
89
90        app_id = sc.applicationId
91
92        print('app end: ', app_id)
93
94        sc.stop()
95        return app_id
96      except:
97        print("Configuration error: "+str(internal_param))
98        sc.stop()
99
100   def word_count_level_2_mem_and_disk_frequencies(internal_param, data_file):
101     try:
102       conf = SparkConf().setMaster("spark://dana:7077").setAppName(internal_param[1]).
              setAll([('spark.driver.cores', internal_param[2]), ('spark.driver.memory',
              internal_param[3]), ('spark.executor.instances', internal_param[4]), ('spark.
              executor.memory', internal_param[5]), ('spark.executor.cores', internal_param
              [6])])
103       sc = SparkContext(conf=conf, pyFiles=['run_app.py', 'persist_dic.py', 'run_job.py'
              ])
104
105       data = sc.textFile(data_file)
106       words = data.flatMap(cleanData)
107       frequencies= words.filter(lambda x: x != '').map(lambda x: (x, 1)).reduceByKey(
              lambda a,b: a+b)
108       frequencies.persist(StorageLevel(True, True, False, False))
109
110       initialsFreqs = frequencies.map(lambda x: (x[0][0], (x[0], x[1])))
111       groupInitialFreqs = initialsFreqs.groupByKey()
112       print('Example of words frequencies group by initial:', groupInitialFreqs.collect
              ()[1])
113
114       app_id = sc.applicationId
115
116       print('app end: ', app_id)
117
118       sc.stop()
119       return app_id
120     except:
121       print("Configuration error: "+str(internal_param))
122       sc.stop()
```

```
123
124   def word_count_level_2_disk_only_frequencies(internal_param, data_file):
125     try:
126       conf = SparkConf().setMaster("spark://dana:7077").setAppName(internal_param[1]).
              setAll([('spark.driver.cores', internal_param[2]), ('spark.driver.memory',
              internal_param[3]), ('spark.executor.instances', internal_param[4]), ('spark.
              executor.memory', internal_param[5]), ('spark.executor.cores', internal_param
              [6])])
127       sc = SparkContext(conf=conf, pyFiles=['run_app.py', 'persist_dic.py', 'run_job.py'
              ])
128
129       data = sc.textFile(data_file)
130       words = data.flatMap(cleanData)
131       frequencies= words.filter(lambda x: x != '').map(lambda x: (x, 1)).reduceByKey(
              lambda a,b: a+b)
132       frequencies.persist(StorageLevel(True, False, False, False))
133
134       initialsFreqs = frequencies.map(lambda x: (x[0][0], (x[0], x[1])))
135       groupInitialFreqs = initialsFreqs.groupByKey()
136       print('Example of words frequencies group by initial:', groupInitialFreqs.collect
              ()[1])
137
138       app_id = sc.applicationId
139
140       print('app end: ', app_id)
141
142       sc.stop()
143       return app_id
144     except:
145       print("Configuration error: "+str(internal_param))
146       sc.stop()
147
148   def word_count_plus_mem_only_initialsFreqs(internal_param, data_file):
149     try:
150       conf = SparkConf().setMaster("spark://dana:7077").setAppName(internal_param[1]).
              setAll([('spark.driver.cores', internal_param[2]), ('spark.driver.memory',
              internal_param[3]), ('spark.executor.instances', internal_param[4]), ('spark.
              executor.memory', internal_param[5]), ('spark.executor.cores', internal_param
              [6])])
151       sc = SparkContext(conf=conf, pyFiles=['run_app.py', 'persist_dic.py', 'run_job.py'
              ])
152
153       data = sc.textFile(data_file)
154
155       words = data.flatMap(cleanData)
156       frequencies= words.filter(lambda x: x != '').map(lambda x: (x, 1)).reduceByKey(
              lambda a,b: a+b)
157
158       initialsFreqs = frequencies.map(lambda x: (x[0][0], (x[0], x[1])))
159       initialsFreqs.persist(StorageLevel(False, True, False, False))
160       groupInitialFreqs = initialsFreqs.groupByKey()
161
162       reduceInitialFreqs = initialsFreqs.map(lambda x: (x[0], x[1][1])).reduceByKey(
              lambda a,b: a + b)
163       totalInfoFreqs = reduceInitialFreqs.join(initialsFreqs)
164       print('Example of result: ', totalInfoFreqs.collect()[1])
165
166       app_id = sc.applicationId
167       print('app end: ', app_id)
168
169       sc.stop()
170       return app_id
171     except:
172       print("Configuration error: "+str(internal_param))
173       sc.stop()
174
175   def word_count_plus_mem_and_disk_initialsFreqs(internal_param, data_file):
176     try:
177       conf = SparkConf().setMaster("spark://dana:7077").setAppName(internal_param[1]).
              setAll([('spark.driver.cores', internal_param[2]), ('spark.driver.memory',
              internal_param[3]), ('spark.executor.instances', internal_param[4]), ('spark.
              executor.memory', internal_param[5]), ('spark.executor.cores', internal_param
              [6])])
178       sc = SparkContext(conf=conf, pyFiles=['run_app.py', 'persist_dic.py', 'run_job.py'
              ])
179
180       data = sc.textFile(data_file)
181
182       words = data.flatMap(cleanData)
183       frequencies= words.filter(lambda x: x != '').map(lambda x: (x, 1)).reduceByKey(
              lambda a,b: a+b)
184
185       initialsFreqs = frequencies.map(lambda x: (x[0][0], (x[0], x[1])))
186       initialsFreqs.persist(StorageLevel(True, True, False, False))
187       groupInitialFreqs = initialsFreqs.groupByKey()
188
```

```
189         reduceInitialFreqs = initialsFreqs.map(lambda x: (x[0], x[1][1])).reduceByKey(
                lambda a,b: a + b)
190         totalInfoFreqs = reduceInitialFreqs.join(initialsFreqs)
191         print('Example of result: ', totalInfoFreqs.collect()[1])
192 
193         app_id = sc.applicationId
194         print('app end: ', app_id)
195 
196         sc.stop()
197         return app_id
198       except:
199         print("Configuration error: "+str(internal_param))
200         sc.stop()
201 
202     def word_count_plus_disk_only_initialsFreqs(internal_param, data_file):
203       try:
204         conf = SparkConf().setMaster("spark://dana:7077").setAppName(internal_param[1]).
                setAll([('spark.driver.cores', internal_param[2]), ('spark.driver.memory',
                internal_param[3]), ('spark.executor.instances', internal_param[4]), ('spark.
                executor.memory', internal_param[5]), ('spark.executor.cores', internal_param
                [6])])
205         sc = SparkContext(conf=conf, pyFiles=['run_app.py', 'persist_dic.py', 'run_job.py'
                ])
206 
207         data = sc.textFile(data_file)
208 
209         words = data.flatMap(cleanData)
210         frequencies= words.filter(lambda x: x != '').map(lambda x: (x, 1)).reduceByKey(
                lambda a,b: a+b)
211 
212         initialsFreqs = frequencies.map(lambda x: (x[0][0], (x[0], x[1])))
213         initialsFreqs.persist(StorageLevel(True, False, False, False))
214         groupInitialFreqs = initialsFreqs.groupByKey()
215 
216         reduceInitialFreqs = initialsFreqs.map(lambda x: (x[0], x[1][1])).reduceByKey(
                lambda a,b: a + b)
217         totalInfoFreqs = reduceInitialFreqs.join(initialsFreqs)
218         print('Example of result: ', totalInfoFreqs.collect()[1])
219 
220         app_id = sc.applicationId
221         print('app end: ', app_id)
222 
223         sc.stop()
224         return app_id
225       except:
226         print("Configuration error: "+str(internal_param))
227         sc.stop()
```

shuffle_scripts.py

```
from pyspark import SparkContext, SparkConf
import sys
import string
import re

def cleanData(x):
  x = re.sub('[^a-zA-Z0-9 ]', "", x.lower())
  line = x.split(" ")
  return line

def word_count_level_2_reduce(internal_param, data_file):
  try:
    conf = SparkConf().setMaster("spark://dana:7077").setAppName(internal_param[1]).
        setAll([('spark.driver.cores', internal_param[2]), ('spark.driver.memory',
        internal_param[3]), ('spark.executor.instances', internal_param[4]), ('spark.
        executor.memory', internal_param[5]), ('spark.executor.cores', internal_param
        [6])])
    sc = SparkContext(conf=conf, pyFiles=['run_app.py', 'shuffle_dic.py', 'run_job.py'
        ])

    data = sc.textFile(data_file)
    words = data.flatMap(cleanData)
    frequencies= words.filter(lambda x: x != '').map(lambda x: (x, 1)).reduceByKey(
        lambda a,b: a+b)

    initialsFreqs = frequencies.map(lambda x: (x[0][0], (x[0], x[1])))
    groupInitialFreqs = initialsFreqs.reduceByKey(lambda a,b: a+b)
    print('Example of words frequencies group by initial:', groupInitialFreqs.collect
        ()[1])

    app_id = sc.applicationId

    print('app end: ', app_id)

    sc.stop()
    return app_id
  except:
    print("Configuration error: "+str(internal_param))
    sc.stop()

def word_count_plus_reduce(internal_param, data_file):
  try:
    conf = SparkConf().setMaster("spark://dana:7077").setAppName(internal_param[1]).
        setAll([('spark.driver.cores', internal_param[2]), ('spark.driver.memory',
        internal_param[3]), ('spark.executor.instances', internal_param[4]), ('spark.
        executor.memory', internal_param[5]), ('spark.executor.cores', internal_param
        [6])])
    sc = SparkContext(conf=conf, pyFiles=['run_app.py', 'shuffle_dic.py', 'run_job.py'
        ])

    data = sc.textFile(data_file)
    words = data.flatMap(cleanData)
    frequencies= words.filter(lambda x: x != '').map(lambda x: (x, 1)).reduceByKey(
        lambda a,b: a+b)

    initialsFreqs = frequencies.map(lambda x: (x[0][0], (x[0], x[1])))
    groupInitialFreqs = initialsFreqs.reduceByKey(lambda a,b: a+b)

    reduceInitialFreqs = initialsFreqs.map(lambda x: (x[0], x[1][1])).reduceByKey(
        lambda a,b: a + b)
    totalInfoFreqs = reduceInitialFreqs.join(initialsFreqs)
    print('Example of result: ', totalInfoFreqs.collect()[1])

    app_id = sc.applicationId
    print('app end: ', app_id)

    sc.stop()
```

```
54        return app_id
55      except:
56        print("Configuration error: "+str(internal_param))
57        sc.stop()
58
59    def word_count_level_2_reduce_jobs(internal_param, data_file):
60      try:
61        conf = SparkConf().setMaster("spark://dana:7077").setAppName(internal_param[1]).
              setAll([('spark.driver.cores', internal_param[2]), ('spark.driver.memory',
              internal_param[3]), ('spark.executor.instances', internal_param[4]), ('spark.
              executor.memory', internal_param[5]), ('spark.executor.cores', internal_param
              [6])])
62        sc = SparkContext(conf=conf, pyFiles=['run_app.py', 'shuffle_dic.py', 'run_job.py'
              ])
63
64        data = sc.textFile(data_file)
65        words = data.flatMap(cleanData)
66        frequencies= words.filter(lambda x: x != '').map(lambda x: (x, 1)).reduceByKey(
              lambda a,b: a+b)
67        print('Example of words frequencies:', frequencies.collect()[1])
68
69        initialsFreqs = frequencies.map(lambda x: (x[0][0], (x[0], x[1])))
70        groupInitialFreqs = initialsFreqs.reduceByKey(lambda a,b: a+b)
71        print('Example of words frequencies group by initial: ', groupInitialFreqs.collect
              ()[1])
72
73        app_id = sc.applicationId
74
75        print('app end: ', app_id)
76
77        sc.stop()
78        return app_id
79      except:
80        print("Configuration error: "+str(internal_param))
81        sc.stop()
82
83    def word_count_plus_reduce_jobs(internal_param, data_file):
84      try:
85        conf = SparkConf().setMaster("spark://dana:7077").setAppName(internal_param[1]).
              setAll([('spark.driver.cores', internal_param[2]), ('spark.driver.memory',
              internal_param[3]), ('spark.executor.instances', internal_param[4]), ('spark.
              executor.memory', internal_param[5]), ('spark.executor.cores', internal_param
              [6])])
86        sc = SparkContext(conf=conf, pyFiles=['run_app.py', 'shuffle_dic.py', 'run_job.py'
              ])
87
88        data = sc.textFile(data_file)
89        words = data.flatMap(cleanData)
90        frequencies= words.filter(lambda x: x != '').map(lambda x: (x, 1)).reduceByKey(
              lambda a,b: a+b)
91        print('Example of words frequencies:', frequencies.collect()[1])
92
93        initialsFreqs = frequencies.map(lambda x: (x[0][0], (x[0], x[1])))
94        groupInitialFreqs = initialsFreqs.reduceByKey(lambda a,b: a+b)
95        print('Example of words frequencies group by initial: ', groupInitialFreqs.collect
              ()[1])
96
97        reduceInitialFreqs = initialsFreqs.map(lambda x: (x[0], x[1][1])).reduceByKey(
              lambda a,b: a + b)
98        totalInfoFreqs = reduceInitialFreqs.join(initialsFreqs)
99        print('Example of result: ', totalInfoFreqs.collect()[1])
100
101       app_id = sc.applicationId
102       print('app end: ', app_id)
103
104       sc.stop()
105       return app_id
106     except:
107       print("Configuration error: "+str(internal_param))
108       sc.stop()
```

skew_scripts.py

```
from pyspark import SparkContext, SparkConf
import sys
import string
import re

def cleanData(x):
  x = re.sub('[^a-zA-Z ]', "", x.lower())
  line = x.split(" ")
  return line

def cleanData2(x):
  line = x.split(",")
  return line


def app_join_alph(internal_param, data_file):
  try:
    conf = SparkConf().setMaster("spark://dana:7077").setAppName(internal_param[1]).
        setAll([('spark.driver.cores', internal_param[2]), ('spark.driver.memory',
        internal_param[3]), ('spark.executor.instances', internal_param[4]), ('spark.
        executor.memory', internal_param[5]), ('spark.executor.cores', internal_param
        [6])])
    sc = SparkContext(conf=conf, pyFiles=['run_app.py','skew_dic.py', 'run_job.py'])

    data = sc.textFile(data_file)
    words = data.flatMap(cleanData)

    alphDim = sc.textFile('/user/janira/alphDim.txt').map(cleanData2).map(lambda x: (x
        [0], x[1]))

    frequencies= words.filter(lambda x: x != '').map(lambda x: (x, 1)).reduceByKey(
        lambda a,b: a+b)

    startWith = frequencies.map(lambda x: (x[0][0], (x[0],x[1])))

    startWithFreqs = frequencies.map(lambda x: (x[0][0], x[1]))

    wordsVocalsDim = startWithFreqs.join(vocalsDim)
    print(wordsVocalsDim.collect()[0])

    app_id = sc.applicationId
    sc.stop()
    return app_id
  except:
    print("Configuration error: "+str(internal_param))
    sc.stop()

def app_join_alph_broadcast(internal_param, data_file):
  try:
    conf = SparkConf().setMaster("spark://dana:7077").setAppName(internal_param[1]).
        setAll([('spark.driver.cores', internal_param[2]), ('spark.driver.memory',
        internal_param[3]), ('spark.executor.instances', internal_param[4]), ('spark.
        executor.memory', internal_param[5]), ('spark.executor.cores', internal_param
        [6])])
    sc = SparkContext(conf=conf, pyFiles=['run_app.py','skew_dic.py', 'run_job.py'])

    data = sc.textFile(data_file)
    words = data.flatMap(cleanData)

    vocalsDim = sc.textFile('/user/janira/alphDim.txt').map(cleanData2).map(lambda x:
        (x[0], x[1]))
    vocalsDimBc = sc.broadcast({ vocalsDim.collect()[i][0] : vocalsDim.collect()[i][1]
         for i in range (0,len(vocalsDim.collect())) })

    frequencies= words.filter(lambda x: x != '').map(lambda x: (x, 1)).reduceByKey(
        lambda a,b: a+b)

```

```
55        startWith = frequencies.map(lambda x: (x[0][0], (x[0],x[1])))
56
57        startWithFreqs = frequencies.map(lambda x: (x[0][0], x[1]))
58
59        wordsVocalsDim = startWithFreqs.map(lambda x: (x[0], (x[1], vocalsDimBc.value[x
              [0]])))
60        print(wordsVocalsDim.collect()[0])
61
62        app_id = sc.applicationId
63        sc.stop()
64        return app_id
65      except:
66        print("Configuration error: "+str(internal_param))
67        sc.stop()
```

logscripts.py

```
import sys
import json
from datetime import datetime
import os
import string
import os.path
import shutil
import sys
import ast

def write_results(data_f, internal_param, resultadosFile, totalsFile, file):
    loaded_json = json.dumps(file.readlines())
    log = json.loads(loaded_json)

    appStart = 0
    appEnd = 0
    jobsInfo = []
    stagesInfo = []
    tasksInfo = []
    tasksTimes = []
    JVCGCTimeMetrics = []
    ShuffleRecordsMetrics = []
    ShuffleWriteMetrics = []
    ShuffleBytesWrittenMetrics = []

    internal_param = internal_param.strip('][').split(',')[1:]

    for event in log:
        x = json.loads(event)
        if x["Event"] == "SparkListenerApplicationStart":
            appStart = datetime.utcfromtimestamp(x["Timestamp"]/1000)
            resultadosFile.write("App Start Time: "+str(appStart)+"\n\n")

        elif  x["Event"] == "SparkListenerApplicationEnd":
            appEnd = datetime.utcfromtimestamp(x["Timestamp"]/1000)
            resultadosFile.write("App Finnish Time: "+str(appEnd)+"\n\n")

        elif x["Event"] == "SparkListenerJobStart":
            startTime = datetime.utcfromtimestamp(x["Submission Time"]/1000)
            jobsInfo.append((x["Job ID"], startTime))
            resultadosFile.write("  Job "+ str(x["Job ID"]) + " Start, Number of
                Stages " + str(len(x["Stage IDs"]))+"\n\n")

        elif x["Event"] == "SparkListenerStageSubmitted":
            stagesInfo.append((x["Stage Info"]["Stage ID"], x["Stage Info"]["Number of
                 Tasks"]))
            resultadosFile.write("    Stage "+ str(x["Stage Info"]["Stage ID"]) + "
                Start, Number of Tasks " + str(x["Stage Info"]["Number of Tasks"])+"\n
                \n")

        elif x["Event"] == "SparkListenerTaskStart":
            startTime = datetime.utcfromtimestamp(x["Task Info"]["Launch Time"]/1000)
            tasksInfo.append((x["Task Info"]["Task ID"], startTime))

        elif x["Event"] == "SparkListenerTaskEnd":
            for i in range(len(tasksInfo)):
                if tasksInfo[i][0] == x["Task Info"]["Task ID"]:
                    finnishTime = datetime.utcfromtimestamp(x["Task Info"]["Finish
                        Time"]/1000)
                    taskTime = (finnishTime - tasksInfo[i][1]).total_seconds()
                    tasksTimes.append(taskTime)
            JVCGCTime = x["Task Metrics"]["JVM GC Time"]
            JVCGCTimeMetrics.append(int(JVCGCTime))
            ShuffleRecords = x["Task Metrics"]["Shuffle Write Metrics"]["Shuffle
                Records Written"]
            ShuffleRecordsMetrics.append(int(ShuffleRecords))
```

```
61                ShuffleWrite = x["Task Metrics"]["Shuffle Write Metrics"]["Shuffle Write
                      Time"]
62                ShuffleWriteMetrics.append(int(ShuffleWrite))
63                ShuffleBytesWritten = x["Task Metrics"]["Shuffle Write Metrics"]["Shuffle
                      Bytes Written"]
64                ShuffleBytesWrittenMetrics.append(int(ShuffleBytesWritten))
65
66                resultadosFile.write("      Finnish Task "+ str(x["Task Info"]["Task ID"])
                       +", Total time: "+ str(taskTime)+ "\n")
67                resultadosFile.write("           JVM GC Time "+ str(x["Task Metrics"]["JVM
                       GC Time"])+ "\n")
68                resultadosFile.write("           Shuffle Read Metrics "+ str(x["Task
                      Metrics"]["Shuffle Read Metrics"])+ "\n")
69                resultadosFile.write("           Shuffle Records Written "+ str(x["Task
                      Metrics"]["Shuffle Write Metrics"]["Shuffle Records Written"])+ "\n")
70                resultadosFile.write("           Shuffle Write Time "+ str(x["Task Metrics
                      "]["Shuffle Write Metrics"]["Shuffle Write Time"])+ "\n")
71                resultadosFile.write("           Shuffle Bytes Written "+ str(x["Task
                      Metrics"]["Shuffle Write Metrics"]["Shuffle Bytes Written"])+ "\n")
72                resultadosFile.write("           Input Records Read "+ str(x["Task Metrics
                      "]["Input Metrics"]["Records Read"])+ "\n")
73                resultadosFile.write("           Input Bytes Read "+ str(x["Task Metrics"
                      ]["Input Metrics"]["Bytes Read"])+ "\n")
74                resultadosFile.write("           Output Records Written " + str(x["Task
                      Metrics"]["Output Metrics"]["Records Written"])+ "\n")
75                resultadosFile.write("           Output Bytes Written " + str(x["Task
                      Metrics"]["Output Metrics"]["Bytes Written"])+ "\n\n\n")
76
77            elif x["Event"] == "SparkListenerStageCompleted":
78                resultadosFile.write("    ........................................")
79
80            elif x["Event"] == "SparkListenerJobEnd":
81                for i in range(len(jobsInfo)):
82                    if jobsInfo[i][0] == x["Job ID"]:
83                        finnishTime = datetime.utcfromtimestamp(x["Completion Time"]/1000)
84                        jobTime = (finnishTime - jobsInfo[i][1]).total_seconds()
85                        jobsInfo.append((x["Job ID"], jobTime))
86                        resultadosFile.write("  Finnish Job "+ str(x["Job ID"]) + ", Total
                              time: "+ str(jobTime)+"\n\n")
87
88        for i in range(len(stagesInfo)):
89            resultadosFile.write("    Stage "+str(stagesInfo[i][0])+ " completed info:")
90            resultadosFile.write("\n")
91
92            k=0
93            if i != 0:
94                k = k + int(stagesInfo[i-1][1])
95
96            tasks = tasksTimes[k:int(stagesInfo[i][1])+k]
97            resultadosFile.write("      Max task time: " + str(max(tasks))+ ", task ID: "+
                  str(tasks.index(max(tasks))+k)+"\n")
98            resultadosFile.write("      Min task time: " + str(min(tasks))+ ", task ID: "+
                  str(tasks.index(min(tasks))+k)+"\n")
99            mediumTime = sum(tasks)/int(stagesInfo[i][1])
100           resultadosFile.write("      Medium time of tasks: "+ str(mediumTime))
101           totalsFile.write(str(data_f)+str(internal_param[0:])+"MediumTaskTime-Stage"+
                  str(stagesInfo[i][0])+";"+str(mediumTime)+"\n")
102
103           JVCGCTimeMetricsStage = JVCGCTimeMetrics[k:int(stagesInfo[i][1])+k]
104           mediumJVCGCTimeMetrics = sum(JVCGCTimeMetricsStage)/int(stagesInfo[i][1])
105           resultadosFile.write("      Medium JVCGCTimeMetrics of tasks: "+ str(
                  mediumJVCGCTimeMetrics))
106           totalsFile.write(str(data_f)+str(internal_param[0:])+"
                  MediumTaskJVCGCTimeMetrics-Stage"+str(stagesInfo[i][0])+";"+str(
                  mediumJVCGCTimeMetrics)+"\n")
107
108           ShuffleRecordsMetricsStage = ShuffleRecordsMetrics[k:int(stagesInfo[i][1])+k]
109           mediumShuffleRecordsMetrics = sum(ShuffleRecordsMetricsStage)/int(stagesInfo[i
                  ][1])
110           resultadosFile.write("      Medium ShuffleRecordsMetrics of tasks: "+ str(
                  mediumShuffleRecordsMetrics))
111           totalsFile.write(str(data_f)+str(internal_param[0:])+"
                  MediumTaskShuffleRecordsMetrics-Stage"+str(stagesInfo[i][0])+";"+str(
                  mediumShuffleRecordsMetrics)+"\n")
112
113           ShuffleWriteMetricsStage = ShuffleWriteMetrics[k:int(stagesInfo[i][1])+k]
114           mediumShuffleWriteMetrics = sum(ShuffleWriteMetricsStage)/int(stagesInfo[i
                  ][1])
115           resultadosFile.write("      Medium ShuffleWriteMetrics of tasks: "+ str(
                  mediumShuffleWriteMetrics))
116           totalsFile.write(str(data_f)+str(internal_param[0:])+"
                  MediumTaskShuffleWriteMetrics-Stage"+str(stagesInfo[i][0])+";"+str(
                  mediumShuffleWriteMetrics)+"\n")
117
118           ShuffleBytesWrittenMetricsStage = ShuffleBytesWrittenMetrics[k:int(stagesInfo[
                  i][1])+k]
```

```
119             mediumShuffleBytesWritten = sum(ShuffleBytesWrittenMetricsStage)/int(
                    stagesInfo[i][1])
120             resultadosFile.write("      Medium ShuffleBytesWritten of tasks: "+ str(
                    mediumShuffleBytesWritten))
121             totalsFile.write(str(data_f)+str(internal_param[0:])+"
                    MediumTaskShuffleBytesWritten-Stage"+str(stagesInfo[i][0])+";"+str(
                    mediumShuffleBytesWritten)+"\n")
122
123         resultadosFile.write("\nApp Total Time " + str((appEnd - appStart).total_seconds()
                ) + " in seconds, " + str((appEnd - appStart).total_seconds()/60))
124         resultadosFile.write("\n\n\n\n\n")
125         resultadosFile.write("---------------------------------\n")
126         totalsFile.write(str(data_f)+str(internal_param[0:])+"APPTotalTime;"+str((appEnd -
                appStart).total_seconds())+"\n")
127
128     if __name__ == '__main__':
129
130         if 'repartition' in sys.argv[5].replace(',', ''):
131           args = sys.argv[1]+";"+sys.argv[2]+";"+sys.argv[3]+";"+str(sys.argv[4])+str(sys.
                    argv[5])+str(sys.argv[6])+str(sys.argv[7])+str(sys.argv[8])+str(sys.argv[9])
                    +str(sys.argv[10])+str(sys.argv[11])+";"+str(sys.argv[12])
132
133         else:
134           args = sys.argv[1]+";"+sys.argv[2]+";"+sys.argv[3]+";"+str(sys.argv[4])+str(sys.
                    argv[5])+str(sys.argv[6])+str(sys.argv[7])+str(sys.argv[8])+str(sys.argv[9])
                    +str(sys.argv[10])+";"+str(sys.argv[11])
135
136         argList = args.split(';')
137         app_id = argList[0]
138         log_dir = argList[1]
139         data_f = argList[2]
140         internal_param = argList[3]
141         script = argList[4]
142
143         file = open(log_dir+"/"+internal_param.strip('][').split(',')[1]+"_"+app_id)
144         resultadosFile = open("/home/janira/"+str(script)+"Results"+str(data_f)+"/result-"
                +str(internal_param.strip('][').split(',')[1:])+".txt","a")
145         resultadosFileDf = open("/home/janira/resultsDF"+str(data_f)+"/result-"+str(
                internal_param.strip('][').split(',')[1:])+".txt","a")
146
147         totalsFileCount = open("/home/janira/"+str(script)+"Results"+str(data_f)+"/
                totalsCount.txt","a")
148         totalsFileLevel2 = open("/home/janira/"+str(script)+"Results"+str(data_f)+"/
                totalsLevel2.txt","a")
149         totalsFilePlus = open("/home/janira/"+str(script)+"Results"+str(data_f)+"/
                totalsPlus.txt","a")
150         totalsFileDf = open("/home/janira/resultsDF"+str(data_f)+"/totalsDf.txt","a")
151
152         if "df" in internal_param.strip('][').split(',')[1]:
153           write_results(data_f, internal_param, resultadosFileDf, totalsFileDf, file)
154         elif "plus" in internal_param.strip('][').split(',')[1]:
155           write_results(data_f, internal_param, resultadosFile, totalsFilePlus, file)
156         elif "level_2" in internal_param.strip('][').split(',')[1]:
157           write_results(data_f, internal_param, resultadosFile, totalsFileLevel2, file)
158         else:
159           write_results(data_f, internal_param, resultadosFile, totalsFileCount, file)
```

totals.py

```
import sys
from pyspark import SparkContext, SparkConf
import os

def main(path, file, script):

    data_file = (path+file).replace('/','_')
    os.system('hdfs dfs -mkdir /user/janira/'+path.replace('/','_'))
    os.system('hdfs dfs -rm -f /user/janira/'+path.replace('/','_')+'/'+file[0:6].
        replace('totals','resultados')+path+file)
    os.system('hdfs dfs -put -f /home/janira/'+file[0:6].replace('totals',str(script)+
        'Results')+path+file+' /user/janira/'+path.replace('/','_')+'/')
    os.system('rm '+'/home/janira/'+file[0:6].replace('totals',str(script)+'Results')+
        path+"mediumTotalTimes"+file)
    os.system('rm '+'/home/janira/'+file[0:6].replace('totals',str(script)+'Results')+
        path+"mediumTotalTimesWithShuffleInfo"+file)

    resultadosFile = open('/home/janira/'+file[0:6].replace('totals',str(script)+'
        Results')+path+'mediumTotalTimes'+file,"w")
    resultadosFileShuffle = open('/home/janira/'+file[0:6].replace('totals',str(script
        )+'Results')+path+'mediumTotalTimesWithShuffleInfo'+file,"w")

    sc = SparkContext()
    data = sc.textFile('/user/janira/'+path.replace('/','_')+'/'+file)

    execTime = data.map(lambda x: x.replace('word_count', 'word_count_a') if '
        word_count' == x.split('[')[1].split(',')[0] else x).map(lambda x: x.replace('
        persist', 'a_persist') if 'word_count_persist' in x.split('[')[1].split(',')
        [0] else x).map(lambda x: x.replace('repartition', 'a_repartition') if '
        word_count_repartition' in x.split('[')[1].split(',')[0] else x).map(lambda x:
         ((x.split(";")[0]).strip(), [float(x.split(";")[1]), 1]))
    grupedExecTime = execTime.reduceByKey(lambda a,b: (a[0]+b[0], a[1]+b[1])).map(
        lambda x: (x[0], x[1][0]/x[1][1]))

    for i in grupedExecTime.filter(lambda x: x[0].split(']')[1].split('-')[0] in ('
        APPTotalTime','MediumTaskTime')).sortBy(lambda x: x[0].split(',')[1].split('_'
        )[0:2]).sortBy(lambda x: x[1]).sortBy(lambda x: x[0].split(']')[1].split('-')
        [0]).collect():
      resultadosFile.write(str(i)+"\n")

    for i in grupedExecTime.sortBy(lambda x: x[1]).sortBy(lambda x: x[0].split(']')
        [1].split('-')[0]).sortBy(lambda x: x[0].split('[')[1].split(',')[0].split('_'
        )[0:2]).collect():
      resultadosFileShuffle.write(str(i)+"\n")

if __name__ == '__main__':

    path = str(sys.argv[1])+'/'
    file = str(sys.argv[2])
    script = str(sys.argv[3])
    main(path, file, script)
```

Bibliografía

- White, T. (2015). *Hadoop. The Definitive Guide* O'Reilly Media.
- Konwinski, A. & Karau, H. & Matei Z. & Wendell P. (2015). *Learning Spark Lightning. Fast Big Data Analysis* O'Reilly Media.
- Drabas, T. & Lee, D. (2017). *Learning PySpark* Packt.
- Chambers, B. & Zaharia, M. (2018). *Spark. The Definitive Guide. Big Data Processing Made Simple* O'Reilly Media.
- The Apache Software Foundation (mayo de 2020) *Apache Spark* `https://spark.apache.org/docs/2.3.0/`
- Databrick (enero de 2020) *Apache Spark* `https://databricks.com/spark/about`
- Grover, M. & Malaska, T. (16 junio 2016) *Top 5 Mistakes When Writing Spark Applications* `https://databricks.com/session/top-5-mistakes-when-writing-spark-applications`
- Kozlowski N. (2 de noviembre de 2017) *Partitioning in Apache Spark* `https://medium.com/parrot-prediction/partitioning-in-apache-spark-8134ad840b0`
- Zvara, Z. (2016). *Handling data skew adaptively in Spark using Dynamic Repartitioning* MTA SZTAKI.
- Karau, H. & Warren, R. (2016). *High Performance Spark: Best practices for scaling and optimizing Apache Spark* O'Reilly Media.
- The Apache Software Foundation (abril de 2020) *Tuning Spark* `https://spark.apache.org/docs/2.3.0/tuning`